



华章程序员书库

The Pragmatic Programmers



ANTLR 4 权威指南

ANTLR 4 权威指南

The Definitive ANTLR 4 Reference



[美] 特伦斯·帕尔 著

[美] 特伦斯·帕尔 (Terence Parr) 著
张博 译 孙岚 石寒舟 审校

机械工业出版社
China Machine Press

机械工业出版社
China Machine Press

目录

[封面](#)

[译者序](#)

[前言](#)

[致谢](#)

[第一部分 ANTLR和计算机语言简介](#)

[第1章 初识ANTLR](#)

[1.1 安装ANTLR](#)

[1.2 运行ANTLR并测试识别程序](#)

[第2章 纵观全局](#)

[2.1 从ANTLR元语言开始](#)

[2.2 实现一个语法分析器](#)

[2.3 你再也不能往核反应堆多加水了](#)

[2.4 使用语法分析树来构建语言类应用程序](#)

[2.5 语法分析树监听器和访问器](#)

[第3章 入门的ANTLR项目](#)

[3.1 ANTLR工具、运行库以及自动生成的代码](#)

[3.2 测试生成的语法分析器](#)

[3.3 将生成的语法分析器与Java程序集成](#)

[3.4 构建一个语言类应用程序](#)

[第4章 快速指南](#)

[4.1 匹配算术表达式的语言](#)

[4.2 利用访问器构建一个计算器](#)

[4.3 利用监听器构建一个翻译程序](#)

[4.4 定制语法分析过程](#)

[4.5 神奇的词法分析特性](#)

[第二部分 ANTLR开发语言类应用程序](#)

[第5章 设计语法](#)

[5.1 从编程语言的范例代码中提取语法](#)

[5.2 以现有的语法规则为指南](#)

[5.3 使用ANTLR语法识别常见的语言模式](#)

[5.4 处理优先级、左递归和结合性](#)

[5.5 识别常见的词法结构](#)

[5.6 划定词法分析器和语法分析器的界线](#)

[第6章 探索真实的语法世界](#)

[6.1 解析CSV文件](#)

[6.2 解析JSON](#)

[6.3 解析DOT语言](#)

[6.4 解析Cymbol语言](#)

[6.5 解析R语言](#)

[第7章 将语法和程序的逻辑代码解耦](#)

[7.1 从内嵌动作到监听器的演进](#)

[7.2 使用语法分析树监听器编写程序](#)

[7.3 使用访问器编写程序](#)

[7.4 标记备选分支以获取精确的事件方法](#)

[7.5 在事件方法中共享信息](#)

[第8章 构建真实的语言类应用程序](#)

[8.1 加载CSV数据](#)

[8.2 将JSON翻译成XML](#)

[8.3 生成调用图](#)

[8.4 验证程序中符号的使用](#)

[第三部分 高级特性](#)

[第9章 错误报告与恢复](#)

[9.1 错误处理入门](#)

[9.2 修改和转发ANTLR的错误消息](#)

[9.3 自动错误恢复机制](#)

[9.4 勘误备选分支](#)

[9.5 修改ANTLR的错误处理策略](#)

[第10章 属性和动作](#)

[10.1 使用带动作的语法编写一个计算器](#)

[10.2 访问词法符号和规则的属性](#)

[10.3 识别关键字不固定的语言](#)

[第11章 使用语义判定修改语法分析过程](#)

[11.1 识别编程语言的多种方言](#)

[11.2 关闭词法符号](#)

[11.3 识别歧义性文本](#)

[第12章 掌握词法分析的“黑魔法”](#)

[12.1 将词法符号送入不同通道](#)

[12.2 上下文相关的词法问题](#)

[12.3 字符流中的孤岛](#)

[12.4 对XML进行语法分析和词法分析](#)

[第四部分 ANTLR参考文档](#)

[第13章 探究运行时API](#)

[13.1 包结构概览](#)

[13.2 识别器](#)

[13.3 输入字符流和词法符号流](#)

[13.4 词法符号和词法符号工厂](#)

[13.5 语法分析树](#)

[13.6 错误监听器和监听策略](#)

[13.7 提高语法分析器的速度](#)

[13.8 无缓冲的字符流和词法符号流](#)

[13.9 修改ANTLR的代码生成机制](#)

[第14章 移除直接左递归](#)

[14.1 直接左递归备选分支模式](#)

[14.2 左递归规则转换](#)

[第15章 语法参考](#)

[15.1 语法词汇表](#)

[15.2 语法结构](#)

[15.3 文法规则](#)

[15.4 动作和属性](#)

[15.5 词法规则](#)

[15.6 通配符与非贪婪子规则](#)

[15.7 语义判定](#)

[15.8 选项](#)

[15.9 ANTLR命令行参数](#)

[参考文献](#)

本书由“ePUBw.COM”整理, ePUBw.COM 提供
最新最全的优质电子书下载!!!

封面

The
Pragmatic
Programmers



ANTLR 4

权威指南

The Definitive ANTLR 4 Reference



[美] 特恩斯·帕尔 (Terence Parr) 著

张博 译 孙岚 石寒舟 审校



机械工业出版社
China Machine Press

本书由“ePUBw.COM”整理，ePUBw.COM 提供
最新最全的优质电子书下载！！！！

译者序

四年前，我在读研究生时曾经参考龙书编写过一个简单的编译器前端。经过一个星期的实践后，我意识到，从头实现一个编译器前端的难度远远超出了一般开发者的能力。编写编译器前端所需要的理论基础、技术功底和精力都远非普通软件可比。

幸运的是，ANTLR的出现使这个过程变得易如反掌。ANTLR能够根据用户定义的语法文件自动生成词法分析器和语法分析器，并将输入文本处理为（可视化的）语法分析树。这一切都是自动进行的，所需的仅仅是一份描述该语言的语法文件。

一年前，我在为淘宝的一个内部数据分析系统设计DSL时，第一次接触到了ANTLR。使用ANTLR之后，我在一天之内就完成了整个编译器前端的开发工作，从而能够迅速开始处理真正的业务逻辑。从那时起，我就被它强大的功能所深深吸引。简而言之，ANTLR能够解决别的工具无法解决的问题。

软件改变了世界。数十年来，信息化的浪潮在全球颠覆着一个又一个的行业。然而，整个世界的信息化程度还远未达到合理的高度，还有大量传统行业的生产力可以被信息化所解放。在这种看似矛盾的情形背后存在着一条鸿沟：大量从事传统行业的人员拥有在本行业中无与伦比的业务知识和经验，却苦于跟不上现代软件发展的脚步。解决这个问题的根本方法就是DSL（Domain Specific Language），让传统行业的人员能够用严谨的方式与计算机对话。其实，本质上任何编程语言都是一种DSL，殊途同归。

而实现DSL的主要困难就在编译器前端。编译器被称为软件工程皇冠上的明珠。一直以来，对于普通的开发者而言，编译器的设计与实现都如同诗中描述的那样：“白云在青天，可望不可即。”

ANTLR改变了这一切。ANTLR自动生成的编译器前端高效、准确，能够将开发者从繁杂的编译理论中解放出来，集中精力处理自己的业务逻辑。ANTLR 4引入的自动语法分析树创建与遍历机制，极大地提高了语言识别程序的开发效率。

时至今日，ANTLR仍然是Java世界中实现编译器的不二之选，同时，它对其他编程语言也提供了不同程度的支持。在开始学习ANTLR时，我发现国内有关ANTLR的资料较为贫乏，这催生了我翻译本书的念头。我期望通过本书的翻译，让更多的开发者能够更加自如地解决职业生涯中碰到的难题。

本书没有冗长的理论，而是从一些具体的需求出发，由浅入深地介绍了语言的背景知识、ANTLR语法的设计方法以及基于ANTLR 4实现语言识别程序的详细步骤。它尤其适用于对语言识别程序的开发感兴趣的开发者。不过，假如你现在没有这样的需求，我仍然建议你阅读本书，因为它能够开拓你的眼界，让你深入实现层面加深对编程语言的理解。

感谢原作者Terence Parr教授向这个世界贡献了如此优秀的软件。您编写的ANTLR极大地提高了开发效率，这实际上等于延长了广大开发者的生命。

感谢孙岚和石寒舟两位前辈对本书审校付出的心血，您二位的宝贵建议令我受益匪浅。

感谢华章公司的和静编辑对本书的翻译提供的支持与帮助。

感谢我的妻子张洁珊女士，你的理解和陪伴保障了翻译过程如期完成。

感谢每一位读者，你的潜心研习与融会贯通将会令本书更有价值。

截止本书译完的2016年12月，ANTLR已经演进到了4.6。在这个过程中，一些Breaking Change出现了，本书中的部分示例代码已经不再有效。因此，我尽自己所能，结合勘误表，使用最新版的ANTLR对它们

进行了逐个验证。对于失效的代码，我通过译注的方式予以修正。由于译者水平有限，书中出现错误与不妥之处在所难免，恳请读者批评指正。

张博

2017年1月

本书由“ePUBw.COM”整理，ePUBw.COM 提供
最新最全的优质电子书下载！！！！

前言

ANTLR是一款强大的语法分析器生成工具，可用于读取、处理、执行和翻译结构化的文本或二进制文件。它被广泛应用于学术领域和工业生产实践，是众多语言、工具和框架的基石。Twitter搜索使用ANTLR进行语法分析，每天处理超过20亿次查询；Hadoop生态系统中的Hive、Pig、数据仓库和分析系统所使用的语言都用到了ANTLR；Lex Machina将ANTLR用于分析法律文本；Oracle公司在SQL开发者IDE和迁移工具中使用了ANTLR；NetBeans公司的IDE使用ANTLR来解析C++；Hibernate对象-关系映射框架（ORM）使用ANTLR来处理HQL语言。

除了这些鼎鼎大名的项目之外，还可以利用**ANTLR**构建各种各样的实用工具，如配置文件读取器、遗留代码转换器、维基文本渲染器，以及**JSON**解析器。我编写了一些工具，用于创建数据库的对象-关系映射、描述三维可视化以及在**Java**源代码中插入性能监控代码。我甚至为一次演讲编写了一个简单的**DNA**模式匹配程序。

一门语言的正式描述称为语法（**grammar**），**ANTLR**能够为该语言生成一个语法分析器，并自动建立语法分析树——一种描述语法与输入文本匹配关系的数据结构。**ANTLR**也能够自动生成树的遍历器，这样你就可以访问树中的节点，执行自定义的业务逻辑代码。

本书既是**ANTLR 4**的参考手册，也是解决语言识别问题的指南。你会学到如下知识：

- 识别语言样例和参考手册中的语法模式，从而编写自定义的语法。
- 循序渐进地为从简单的**JSON**到复杂的**R**语言编写语法。同时还能学会解决**XML**和**Python**中棘手的识别问题。
- 基于语法，通过遍历自动生成的语法分析树，实现自己的语言类应用程序。
- 在特定的应用领域中，自定义识别过程的错误处理机制和错误报告机制。

·通过在语法中嵌入Java动作（action），对语法分析过程进行完全的掌控。

本书并非教科书，所有的讨论都是基于实例的，旨在令你巩固所学的知识，并提供语言类应用程序的基本范例。

本书的读者对象

本书尤其适用于对数据读取器、语言解释器和翻译器感兴趣的开发者。虽然本书主要利用ANTLR来完成这些工作，你仍然可以学到很多有关词法分析器和语法分析器的知识。初学者和专家都需要本书来高效地使用ANTLR 4。如果希望学习第三部分中的高级特性，你需要先了解之前章节中的ANTLR基础知识。此外，读者还需要具备一定的Java功底。

Honey Badger版本

ANTLR 4的版本代号是“Honey Badger”，这个名字来源于一段著名的YouTube短片The Crazy Nastyass Honey Badger（网址为：<http://www.youtube.com/watch?v=4r7wHMg5Yjg>）中的勇敢无畏的主角——一只蜜獾。它敢吃你给它的任何东西，根本不在乎那是什么！

ANTLR 4有哪些神奇之处

ANTLR 4引入了一些新功能，降低了入门门槛，使得语法和语言类应用程序的开发更加容易。最重要的新特性在于，ANTLR 4几乎能够处理任何语法（除了间接左递归，稍后会提到）。在ANTLR将你的语法转换成可执行的、人类可读的语法分析代码的过程中，语法冲突或者歧义性警告不会再出现。

无论多复杂的语法，只要你提供给ANTLR自动生成的语法分析器的输入是合法的，该语法分析器就能够自动识别之。当然，你需要自行保证该语法能够准确地描述目标语言。

ANTLR语法分析器使用了一种名为自适应LL (*) 或者ALL (*)（读作“all star”）的新技术，它是由我和Sam Harwell一起开发的。ALL

(*) 是ANTLR 3中的LL (*) 的扩展，在实际生成的语法分析器执行前，它能够在运行时以动态方式对语法执行分析，而非先前的静态方式。由于ALL (*) 语法分析器能够访问实际的输入文本，通过反复分析语法的方式，它最终能够决定如何识别输入文本。相比之下，静态分析必须考虑所有可行的（无限长的）输入序列。

在实践中，拥有ALL (*) 意味着你无须像在其他语法分析器生成工具（包括ANTLR 3）中那样，扭曲语法以适应底层的语法分析策略。如果你曾经为ANTLR 3的歧义性警告和yacc的归约/归约冲突（reduce/reduce conflict）而抓狂，ANTLR 4就是你的不二之选！

另外一个强大的新功能是**ANTLR 4**极大地简化了匹配某些句法结构（如编程语言中的算术表达式）所需的语法规则。长久以来，处理表达式都是**ANTLR**语法（以及手工编写的递归下降语法分析器）的难题。识别表达式最自然的语法对于传统的自顶向下的语法分析器生成器（如**ANTLR 3**）是无效的。现在，利用**ANTLR 4**，你可以通过如下规则匹配表达式：

```
expr : expr '*' expr // 匹配乘号连接的子表达式
      | expr '+' expr // 匹配加号连接的子表达式
      | INT           // 匹配简单的整数因子
      ;
```

类似**expr**的自引用规则是递归的，更准确地说，是左递归（**left recursive**）的，因为它的至少一个备选分支直接引用了它自己。

ANTLR 4自动将类似**expr**的左递归规则重写成了等价的非左递归形式。唯一的约束是左递归必须是直接的，也就是说规则直接引用自身。一条规则不能引用另外一条规则，如果后者的备选分支之一在左侧直接引用了前者（而没有匹配一个词法符号）。详见5.4节。

除了上述两项与语法相关的改进，**ANTLR 4**还使得编写语言类应用程序更加容易。**ANTLR**生成的语法分析器能够自动建立名为语法分析树（**parse tree**）的视图，其他程序可以遍历此树，并在所需处理的结构处触发回调函数。在先前的**ANTLR 3**中，用户需要补充语法来创建树。除了自动建立树结构之外，**ANTLR 4**还能自动生成语法分析树遍历器

的实现：监听器（**listener**）或者访问器（**visitor**）。监听器与在XML文档的解析过程中响应SAX事件的处理器相似。

由于拥有以下几点ANTLR 3所不具备的新特性，ANTLR 4显得非常容易上手：

- 最大的改变是ANTLR 4降低了语法中内嵌动作（代码）的重要性，取而代之的是监听器和访问器。新机制将语法和应用的逻辑代码解耦，使得应用程序本身被封装起来，而非散落在语法的各处。在没有内嵌动作的情况下，你可以在多个程序中复用同一份语法，甚至都无须重新编译生成的语法分析器。虽然ANTLR仍然允许内嵌动作的存在，但是在ANTLR 4中，它们更像是一种进阶用法。这样的行为能够最大程度地掌控语法分析过程，但其代价是语法复用性的丧失。

- 由于ANTLR能够自动生成语法分析树和树的遍历器，在ANTLR 4中，你无须再编写树语法。取而代之的是一些广为人知的设计模式，如访问者模式。这意味着，在学会了ANTLR语法之后，你就可以重回自己熟悉的Java领域来实现真正的语言类应用程序。

- ANTLR 3的LL（*）语法分析策略不如ANTLR 4的ALL（*）强大，所以ANTLR 3为了能够正确识别输入的文本，有时候不得不进行回溯。回溯的存在使得语法的调试格外困难，因为生成的语法分析器会对同

样的输入进行（递归的）多趟语法分析。回溯也为语法分析器在面对非法输入时给出错误消息设置了重重障碍。

ANTLR 4是25年前我读研究生时所走的一小段弯路的成果。我想，我也许会稍微改变我曾经的座右铭。

为什么不花5天时间编程，来使你25年的生活自动化呢？

ANTLR 4正是我所期望的语法分析器生成器，现在，我终于能够回头去研究我原先在20世纪80年代试图解决的问题——假如我还记得它的话。

本书的主要内容

本书是你所能找到的有关ANTLR 4的信息源中最好、最完整的。免费的在线文档提供了足够多有关基础语法的句法和语义的资料，不过没有详细解释ANTLR的相关概念。在本书中，识别语言的语法模式和将其表述为ANTLR语法的内容是独一无二的。贯穿全书的示例能够在构建语言类应用程序方面助你一臂之力。本书可帮助你融会贯通，成为ANTLR专家。

本书由四部分组成。

·第一部分介绍了ANTLR，提供了一些与语言相关的背景知识，并展示了ANTLR的一些简单应用。在这一部分中，你会了解ANTLR的句法以

及主要用途。

- 第二部分是一部有关设计语法和使用语法来构建语言类应用程序的“百科全书”。

- 第三部分展示了自定义ANTLR生成的语法分析器的错误处理机制的方法。随后，你会学到在语法中嵌入动作的方法——在某些场景下，这样做比建立树并遍历之更简单，也更有效率。此外，你还将学会使用语义判定（**semantic predicate**）来修改语法分析器的行为，以便解决一些充满挑战的识别难题。

- 本部分的最后一章解决了一些充满挑战的识别难题，例如识别XML和Python中的上下文相关的换行符。

- 第四部分是参考章节，详细列出了ANTLR语法元语言的所有规则和ANTLR运行库的用法。

完全不了解语法和语言识别工具的读者请务必从头开始阅读。具备ANTLR 3使用经验的用户可从第4章开始阅读以学习ANTLR 4的新功能。

有关ANTLR的更多在线学习资料

在<http://wwwantlr.org>上，你可以找到ANTLR、ANTLRWorks2图形界面开发环境、文档、预制的语法、示例、文章，以及文件共享区。技术

支持邮件组是一个对初学者十分友好的公开讨论组。

Terence Parr

2012年11月于旧金山大学

本书由“ePUBw.COM”整理，ePUBw.COM 提供
最新最全的优质电子书下载！！！！

致谢

大约25年前，我开始致力于ANTLR的相关工作。那时，在许多人的帮助下，ANTLR工具的句法和功能逐渐成形，在此，我向他们致以由衷的感谢。要特别感谢的是Sam Harwell，他是ANTLR 4的另一位开发者。他不仅帮助我完成了此软件，而且在ALL（*）语法分析算法上做出了突出的贡献。Sam也是ANTLRWorks2语法IDE的开发者。

感谢以下人员对本书进行了技术审阅：Oliver Ziegemann、Sam Rose、Kyle Ferrio、Maik Schmidt、Colin Yates、Ian Dees、Tim Ottinger、Kevin Gisi、Charley Stran、Jerry Kuch、Aaron Kalair、Michael Bevilacqua-Linn、Javier Collado、Stephen Wolff以及Bernard Kaiflin。同时，我还要感谢那些在本书和ANTLR 4软件处于beta版本时报告问题的

热心读者。尤其要感谢的是Kim Shrier和Graham Wideman，他们二位的审阅格外认真。Graham的审阅报告之仔细、翔实和广博，令我不知是该紧握他的手予以感谢，还是该为自己的疏漏羞愧难当。

最后，我还要感谢编辑Susannah Davidson Pfalzer，她一如既往地支持我完成了三本书的创作。她提出的宝贵建议和对本书内容的精雕细琢使本书更加完美。

本书由“ePUBw.COM”整理，ePUBw.COM 提供
最新最全的优质电子书下载！！！！

第一部分 ANTLR和计算机语言简介

在第一部分中，我们会安装ANTLR，尝试通过它来识别一个简单的“hello world”语法，并概览语言类应用程序的开发过程。在此基础上，我们会构造一个语法来识别和翻译形如{1, 2, 3}的花括号中的一列整数。最后，我们将通过一系列的简单语法和程序来快速了解ANTLR的特性。

第1章 初识ANTLR

在本书的第一部分中，我们的目标是大体上知道ANTLR能做什么。除此之外，我们还希望探究语言类应用程序的架构。在概览之后的第2章中，我们将会通过许多真实的例子来循序渐进地、系统性地学习ANTLR。在开始之前，我们需要首先安装ANTLR，然后尝试用它编写一份简单的“hello world”语法。

1.1 安装ANTLR

ANTLR是用Java编写的，因此你需要首先安装Java，哪怕你的目标是使用ANTLR来生成其他语言（如C#和C++）的解析器。（我希望在不远的未来ANTLR可以支持更多语言。）ANTLR运行所需的Java版本为1.6或更高。

为什么本书使用命令行

在整本书中，我们都会使用命令行（shell）来运行ANTLR和构建我们的程序。因为开发者使用的开发环境和操作系统五花八门，因此只有操作系统的shell才是我们公用的“界面”。使用shell也使得开发语言程序的每一个步骤更加清晰和明确。在本书中我将会一直使用Mac OS X作为示例，不过这些示例命令理论上应该能够在任何类UNIX系统的shell中正常工作，同时，在稍作修改后，它们应该能够适用于Windows。

安装ANTLR本身仅仅需要下载最新的jar包（例如antlr-4.0-complete.jar），然后把它放在合适的位置。该jar包包含了运行ANTLR的工具和编译、执行ANTLR产生的识别程序所依赖的全部运行库。它们有何区别呢？简而言之，ANTLR工具将语法文件转换成可以识别该语法文件所描述的语言的程序。例如，给定一个识别JSON的语法，ANTLR工具将会根据该语法生成一个程序，此程序可以通过ANTLR运行库来识别输入的JSON。

上述jar包还包含两个用于提供相关支持的库：一个复杂的树形结构生成库和StringTemplate，这一个用于生成代码和其他结构化文本的优秀的模板引擎。在ANTLR 4.0中，语法本身是通过ANTLR 3来识别的，所以上述完整版的jar包还包含ANTLR的早期版本。

StringTemplate引擎

StringTemplate是一个Java编写的模板引擎，用于生成源代码、网页、电子邮件或者其他任何格式化的输出文本（已经支持C#、Python、Ruby和Scala）。StringTemplate在生成多目标的代码、多站点皮肤和国际化/本地化方面表现尤其出色。它是在jGuru.com的多年开发过程中逐渐成形的。StringTemplate也用于生成网站，以及为ANTLR 3和ANTLR 4的代码生成器提供有力的支持。关于StringTemplate的更多信息详见其帮助页面（<http://www.stringtemplate.org/about.html>）。

你可以通过浏览器从ANTLR的网站下载ANTLR，或者使用命令行工具curl：

```
$ cd /usr/local/lib
$ curl -O http://www.antlr.org/download/antlr-4.0-complete.jar
```

在UNIX上，`/usr/local/lib`非常适于存放jar包。在Windows上，似乎没有标准的存放jar包的目录，因此你可以简单地将它放在项目文件夹下。大多数开发环境要求你将jar包放在你的语言类应用程序的依赖列表中。不需要修改配置脚本或者配置文件之类的东西——你只需要保证Java能够找到这个jar包即可。

因为本书使用的是命令行，你需要担负设置CLASSPATH环境变量的重任。通过设置好的CLASSPATH环境变量，Java就能够找到ANTLR工具和运行库。在UNIX系统上，你可以手动执行以下命令或者将其添加到启动脚本中（对于bash命令行，就是`.bash_profile`）：

```
$ export CLASSPATH=".:usr/local/lib/antlr-4.0-complete.jar:$CLASSPATH"
```

注意，CLASSPATH中的点非常关键，它代表当前目录。没有它，Java编译器和Java虚拟机就无法加载当前目录的class文件。在本书中，所有的编译和测试都是在当前目录中进行的。

有两种方式可以检查ANTLR的安装是否正确，第一种是通过不带参数的ANTLR命令行工具，第二种是通过java-jar来直接运行ANTLR的jar包

或者直接调用org antlr.v4.Tool类。

```
$ java -jar /usr/local/lib/antlr-4.0-complete.jar # 启动org antlr.v4.Tool
ANTLR Parser Generator Version 4.0
-o ____ specify output directory where all output is generated
-lib ____ specify location of .tokens files
...
$ java org antlr.v4.Tool # 启动org antlr.v4.Tool
ANTLR Parser Generator Version 4.0
-o ____ specify output directory where all output is generated
-lib ____ specify location of .tokens files
...
```

每次都手动输入这些java命令是一件令人痛苦的事情，所以最好通过别名（alias）或者shell脚本的方式。本书接下来将会使用名为antlr4的别名，在类UNIX系统上的定义如下：

```
$ alias antlr4='java -jar /usr/local/lib/antlr-4.0-complete.jar'
```

此外，也可以将上述命令写入/usr/local/bin。

```
install/antlr4
#!/bin/sh
java -cp "/usr/local/lib/antlr4-complete.jar:$CLASSPATH" org antlr.v4.Tool $*
```

在Windows上，可以通过如下批处理命令（假设ANTLR的jar包已经被放置在C: \libraries）实现：

```
install/antlr4.bat
java -cp C:\libraries\antlr-4.0-complete.jar;%CLASSPATH% org antlr.v4.Tool %*
```

不管用哪种方法，现在我们可以直接使用antlr4命令了。

```
$ antlr4
ANTLR Parser Generator Version 4.0
-o _____ specify output directory where all output is generated
-lib _____ specify location of .tokens files
...
```

如果你看到了和上面一样的帮助信息，证明一切就绪，可以开始接下来的ANTLR之旅了！

1.2 运行ANTLR并测试识别程序

下面是一个简单的、识别类似hello world和hello parrrt的词组的语法：

```
install/Hello.g4
grammar Hello;          // 定义一个名为 Hello 的语法
r : 'hello' ID ;        // 匹配一个关键字 hello 和一个紧随其后的标识符
ID : [a-z]+ ;           // 匹配小写字母组成的标识符
WS : [ \t\r\n]+ -> skip ; // 忽略空格、Tab、换行以及 \r (Windows)
```

为整洁起见，我们把这个语法文件放到它自己的目录里，如/tmp/test。

接下来对该语法文件运行ANTLR命令并编译生成的结果。

```
$ cd /tmp/test
$ # 下载或复制粘贴上述代码，并将 Hello.g4 放在 /tmp/test 目录下
$ antlr4 Hello.g4 # 使用之前定义过的 antlr4 命令生成语法分析器和词法分析器。
$ ls
Hello.g4          HelloLexer.java    HelloParser.java
Hello.tokens      HelloLexer.tokens
HelloBaseListener.java HelloListener.java
$ javac *.java    # 编译 ANTLR 生成的 java 代码
```

对Hello.g4运行ANTLR工具命令生成了一个由HelloParser.java和HelloLexer.java组成的、可以运行的语法识别程序，不过我们还缺一个main程序来触发这个语言识别的过程。（语法分析器和词法分析器的

介绍详见下一章。) 这就是项目刚开始时的典型过程。在开始构建一个实际的程序之前，你可以先熟悉一下这些不同的语法。无须对每个新的语法都编写一个main程序来测试。

ANTLR在运行库中提供了一个名为TestRig的方便的调试工具。它可以详细列出一个语言类应用程序在匹配输入文本过程中的信息，这些输入文本可以来自文件或者标准输入。TestRig使用Java的反射机制来调用编译后的识别程序。与之前一样，最好通过别名或者批处理文件来调用它。在本书中，我将会使用grun作为别名，你可以使用任何你喜欢的别名。

```
$ alias grun='java org.antlr.v4.runtime.misc.TestRig'⊖
```

(注：在本书翻译时的最新版本（ANTLR 4.6）中，TestRig已经移至org.antlr.v4.gui包。——译者注)

测试组件有点像是main () 方法，接收一个语法名和一个起始规则名作为参数，此外，它还接收众多的参数，通过这些参数我们可以指定输出的内容。假设我们希望显示识别过程中生成的词法符号。词法符号是类似于关键字hello和标识符parrrt的符号。可以通过以下命令启动grun，测试之前的语法：

```

⇒ $ grun Hello r -tokens      # 使用 Hello 语法和 r 规则启动 TestRig
⇒ hello parrt                # 键入要被识别的语句
⇒ EOF                        # 在 UNIX 系统上键入 Ctrl+D 或者 Windows 系统上键入 Ctrl+Z
                              # 来输入文件结束符
◀ [@0,0:4='hello',<1>,1:0]    # 以下三行是 grun 的输出
    [@1,6:10='parrt',<2>,1:6]
    [@2,12:11='<EOF>',<-1>,2:0]

```

首先输入上述grun命令，回车，然后输入hello parrt，回车。这个时候，你必须手动输入文件结束符（end-of-file character）来阻止程序继续读取标准输入，否则，程序将什么都不做，静静等待你的下一步输入。由于grun命令使用了-tokens选项，一旦识别程序读取到全部的输入内容，TestRig就会打印出全部的词法符号的列表。

每行输出代表了一个词法符号，其中包含了该词法符号的全部信息。例如，[@1, 6: 10='parrt', <2>, 1: 6]表明，这个词法符号位于第二个位置（从0开始计数），由输入文本的第6个到第10个位置之间的字符组成（包含第6个和第10个，同样从0开始计数）；包含的文本内容是parrt；词法符号类型是2（即ID）；位于输入文本的第一行、第6个位置处（从0开始计数，tab符号被看作一个字符）。

我们可以很容易地打印出LISP风格文本格式的语法分析树（根节点和子节点在同一行）。

```

⇒ $ grun Hello r -tree
⇒ hello parrt
⇒ EOF
◀ (r hello parrt)

```

要想知道识别程序是如何识别输入文本的，最简单的办法是查看可视化的语法分析树。使用`grun-gui`运行`TestRig`，即`grun Hello r-gui`，将产生如图1-1所示的对话框。

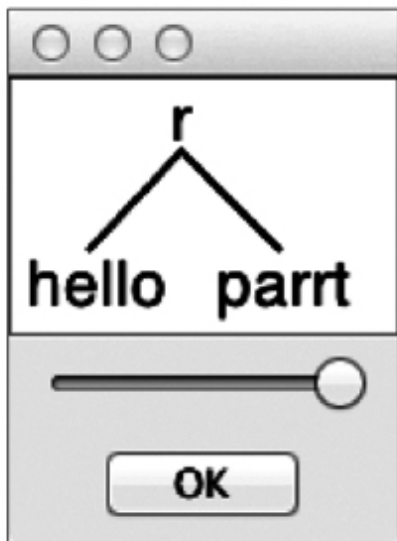


图1-1 运行`TestRig`后的对话框

当不带参数地运行`TestRig`时，会产生一些帮助信息：

```
$ grun
java org.antlr.v4.runtime.misc.TestRig GrammarName startRuleName
  [-tokens] [-tree] [-gui] [-ps file.ps] [-encoding encodingname]
  [-trace] [-diagnostics] [-SLL]
  [input-filename(s)]
Use startRuleName='tokens' if GrammarName is a lexer grammar.
Omitting input-filename makes rig read from stdin.
```

在本书中，我们将会使用其中的很多选项，下面是它们的简单介绍：

-tokens 打印出词法符号流。

-tree 以LISP格式打印出语法分析树。

-gui 在对话框中以可视化方式显示语法分析树。

-ps file.ps 以PostScript格式生成可视化语法分析树，然后将其存储于file.ps。本章中的语法分析树的图片就是使用-ps选项生成的。

-encoding encodingname 若当前的区域设定无法正确读取输入，使用这个选项指定测试组件输入文件的编码。例如，在12.4节中我们需要通过这个选项来解析日语XML文件。

-trace 打印规则的名字以及进入和离开该规则时的词法符号。

-diagnostics 开启解析过程中的调试信息输出。通常仅在一些罕见情况下才使用它产生信息，例如输入的文本有歧义。

-SLL 使用另外一种更快但是功能稍弱的解析策略。

现在，我们已经成功地安装了ANTLR，并尝试着用它分析了一个简单的语法。在下一章中，让我们后退一步，先纵观全局，学习一些重要的术语。之后，我们将会尝试建立一个简单的入门工程来识别和翻译一系列形如{1, 2, 3}的数字。接下来，在第4章中我们将会学习一系列有趣的例子，这些例子展示了ANTLR的强大功能以及可被应用的领域。

第2章 纵观全局

在上一章中，我们安装了ANTLR，了解了如何构建和运行一个简单的示例语法。在本章中，我们将纵观全局，学习语言类应用程序相关的重要过程、术语和数据结构。随着学习的深入，我们将认识一些关键的ANTLR对象，并简单了解ANTLR在背后帮助我们完成的工作。

2.1 从ANTLR元语言开始

为了实现一门编程语言，我们需要构建一个程序，读取输入的语句，对其中的词组和输入符号进行正确的处理。语言（**language**）由一系列有意义的语句组成，语句（**sentence**）由词组组成，词组（**phrase**）是由更小的子词组（**subphrase**）和词汇符号（**vocabulary symbol**）组成。一般来说，如果一个程序能够分析计算或者“执行”语句，我们就称之为解释器（**interpreter**）。这样的例子包括计算器、读取配置文件的程序和Python解释器。如果一个程序能够将一门语言的语句转换为另外一门语言的语句，我们称之为翻译器（**translator**）。这样的例子包括Java到C#的转换器和普通的编译器。

为了达到预期的目的，解释器或者翻译器需要识别出一门特定语言的所有有意义的语句、词组和子词组。识别一个词组意味着我们可以将它从众多的组成部分中辨认和区分出来。例如，我们能够将输入的“**sp=100;**”识别为一个赋值语句，这意味着我们需要知道**sp**是被赋值

的目标，100是要被赋予的值。与之类似，如果我们要识别英文语句，就需要辨认出一段对话的不同部分，例如主语、谓语和宾语。识别语句“sp=100;”还意味着语言类应用程序能够将它和import表达式之类的语句区分开。在成功识别后，程序就能执行适当的操作，诸如performAssignment("sp", 100)或者translateAssignment("sp", 100)。

识别语言的程序称为语法分析器（parser）或者句法分析器（syntax analyzer）。句法（syntax）是指约束语言中的各个组成部分之间关系的规则，在本书中，我们会通过ANTLR语法来指定语言的句法。语法（grammar）是一系列规则的集合，每条规则表述出一种词汇结构。ANTLR工具能够将其转换为如同经验丰富的开发者手工构建一般的语法分析器（ANTLR是一个能够生成其他程序的程序）。ANTLR语法本身又遵循了一种专门用来描述其他语言的语法，我们称之为ANTLR元语言（ANTLR's meta-language）。

如果我们将语法分析的过程分解为两个相似但独立的任务或者说阶段时，实现起来就容易多了。这两个阶段与我们的 brains 阅读英文文本的过程相类似。我们并不是一个字符一个字符地阅读一个句子，而是将句子看作一系列单词。在识别整个句子的语法结构之前，人类的大脑首先通过潜意识将字符聚集为单词，然后获取每个单词的意义。这个过程在阅读摩斯电码的时候更加明显，因为我们需要首先将点和划转换

为字符才能获取消息本身。同样的事情也发生在阅读长单词时，比如说阅读这个单词Humuhumunukunukuapua~~𩚑~~——它是夏威夷的州鱼。

将字符聚集为单词或者符号（词法符号，**token**）的过程称为词法分析（**lexical analysis**）或者词法符号化（**tokenizing**）。我们把可以将输入文本转换为词法符号的程序称为词法分析器（**lexer**）。词法分析器可以将相关的词法符号归类，例如**INT**（整数）、**ID**（标识符）、**FLOAT**（浮点数）等。当语法分析器不关心单个符号，而仅关心符号的类型时，词法分析器就需要将词汇符号归类。词法符号包含至少两部分信息：词法符号的类型（从而能够通过类型来识别词法结构）和该词法符号对应的文本。

第二个阶段是实际的语法分析过程，在这个过程中，输入的词法符号被“消费”以识别语句结构，在上例中即为赋值语句。默认情况下，ANTLR生成的语法分析器会建造一种名为语法分析树（**parse tree**）或者句法树（**syntax tree**）的数据结构，该数据结构记录了语法分析器识别出输入语句结构的过程，以及该结构的各组成部分。图2-1展示了数据在一个语言类应用程序中的基本流动过程。



图2-1 某数据在语言类程序中的流动过程

语法分析树的内部节点是词组名，这些名字用于识别它们的子节点，并将子节点归类。

根节点是最抽象的一个名字，在本例中即stat（statement的简写）。语法分析树的叶子节点永远是输入的词法符号。句子，也即符号的线性组合，本质上是语法分析树在人脑中的串行化。为了能与其他人沟通，我们需要使用一串单词，使得他们能在脑海中构建出一棵相同的语法分析树。

通过语法分析树这种方便的数据结构，语法分析器就能将诸如“符号是如何构成词组的”这样的完整信息传达给程序的其余部分。树结构不仅在后续的步骤中易于处理，而且也是一种为开发者所熟知的数据结构。幸运的是，语法分析器能够自动生成语法分析树。

通过操纵语法分析树，识别同一种语言的不同程序就能复用同一个语法分析器。另外一种解决方案，也是传统的生成语法分析器的方案，是直接在语法文件中嵌入与这种程序相关的代码。ANTLR 4仍然允许这种传统的方案（详见第10章），不过，使用语法分析树可以使程序更整洁、解耦性更强。

在语言的翻译过程中，一个阶段依赖于前一个阶段的计算结果和信息，因此需要多次进行树的遍历（tree walk），这种情况下语法分析树

也是非常有用的。在其他情况下，将一个复杂的程序分解为多个阶段会大大简化编码和测试工作，与其每个阶段都重新解析一下输入的字符流，不如首先生成语法分析树，然后多次访问其中的节点，这样更有效率。

由于我们使用一系列的规则指定语句的词汇结构，语法分析树的子树的根节点就对应语法规则的名字。在下文的长篇大论之前，我们先看一个例子。下面这条语法规则对应图2-1中的赋值语句子树的第一级：

```
assign : ID '=' expr ';' ; // 匹配一个类似 "sp = 100;" 的赋值语句
```

使用和调试ANTLR语法的一个基本要求是，理解ANTLR是如何将这样的规则转换为人类可阅读的语法分析程序的，因此接下来我们将深入研究语法分析的过程。

2.2 实现一个语法分析器

ANTLR工具依据类似于我们之前看到的assign的语法规则，产生一个递归下降的语法分析器（**recursive-descent parser**）。递归下降的语法分析器实际上是若干递归方法的集合，每个方法对应一条规则。下降的过程就是从语法分析树的根节点开始，朝着叶节点（词法符号）进行解析的过程。首先调用的规则，即语义符号的起始点，就会成为语法分析树的根节点。在前一节的例子中，就是调用stat（）方法作为起始点

的。这种解析过程的更广为人知的名字是“自上而下的解析”，递归下降的语法分析器仅仅是自上而下的语法分析器的一种实现。

下面是一个ANTLR根据**assign**规则生成的方法（稍微经过格式整理），用于展示递归下降的语法分析器的实现细节：

```
// assign : ID '=' expr ';' ;
void assign() {      // 根据 assign 规则生成的方法
    match(ID);       // 将当前的输入符号和 ID 相比较，然后将其消费掉
    match('=');
    expr();           // 通过调用方法 expr() 来匹配一个表达式
    match(';');
}
```

递归下降的语法分析器最神奇的地方在于，通过方法**stat ()**、**assign ()**和**expr ()**的调用描绘出的调用路线图映射到了语法分析树的节点上（请迅速回顾一下图2-1）。调用**match ()**对应了语法分析树的叶子节点。在手工构造的语法分析器中，我们需要在每条规则对应的方法的开始位置插入“增加一个新的子树根节点”这样的操作，在**match ()**方法中插入“增加一个新的叶子节点”这样的操作。

assign ()方法仅仅验证所有的词汇符号都存在且顺序正确。当语法分析器进入**assign ()**方法的内部时，仅有一个备选分支（**alternative**），无须做出选择。一个备选分支指的是规则的右侧定义的一个方案之一。例如，除了**assign**之外，下面的**stat**规则还可能对应其他多种语句。

```

/** 从当前输入位置开始，匹配多种语句 */
stat: assign          // 第一个备选分支 ( '|' 符号是备选分支的分隔符 )
    | ifstat          // 第二个备选分支
    | whilestat
    ...
    ;

```

对stat语法规则的解析像是一个switch语句：

```

void stat() {
    switch ( << 当前输入的词法符号 >> ) {
        CASE ID      : assign(); break;
        CASE IF      : ifstat(); break; // IF 是 if 关键字的词法符号类型
        CASE WHILE   : whilestat(); break;
        ...
        default      : << 抛出无可选方案的异常 >>
    }
}

```

stat () 方法必须通过检查下一个词法符号来做出语法分析决策

(parsing decision) 或者预测 (prediction)。做出决策的过程实际上就是判断哪一个备选分支是正确的。在上面的例子中，一个WHILE关键字意味着它选择stat规则的第三个备选分支。因此，stat () 方法将调用whilestat () 方法。你可能听说过前瞻词法符号 (lookahead token) 这个术语，它其实就是下一个输入的词法符号。一个前瞻词法符号是指任何一个在被匹配和消费之前就由语法分析器嗅探出的词法符号。有些时候，语法分析器需要很多个前瞻词法符号来判断语义规则的哪个方案是正确的，甚至可能要从当前的词法符号的位置开始，一直分析到文件末尾才能做出判断！ANTLR默默地帮你完成了所有的这些工作，不过，对其决策过程的基本理解将会有助于调试ANTLR自动生成的语法分析器。

为了让语法分析的决策过程可视化，想象一个迷宫，它只有一个入口和一个出口，迷宫的地板上写着单词。每个从入口到出口的路径上的单词序列代表一个语句。这个迷宫的结构就好比是一种语言所定义的全部语法规则。为了测试一个语句是不是合法，我们将这个语句中的单词和迷宫的地板上的单词比较，然后沿着这个语句的单词所描述的路径在迷宫中前进。如果我们能够通过这个语句中的单词序列指定的路径到达出口，那么这个语句就是合法的。

为了到达迷宫的出口，我们必须在每个分岔路口选择一条正确的路径，就好像一个语法分析器要在多个备选分支中做出选择一样。我们必须将语句中接下来的若干个单词与站在路口所看到的不同岔路地板上的单词相比较，从而决定走哪条岔路。我们站在路口所看到的地板上的单词就好像是前瞻词法符号。显然，若每条岔路都以一个独一无二的单词开始，做出选择就会容易许多。在上例中的`stat`规则中，每个备选分支都是以一个独一无二的词法符号开始的，因此`stat ()`方法可以通过检查第一个前瞻词法符号来区分不同的备选分支。

当每条岔路的起始单词有重复的时候，语法分析器就需要更多地进行前瞻，即通过扫描更多的单词来区分不同的备选分支。在每次语法分析决策中，**ANTLR**能够根据情况自动调整前瞻的数量。如果通过前瞻，我们能够经多条路径抵达迷宫出口（文件末尾），那就意味着能够用多种语义去解释当前的输入文本。解决这种歧义是我们下一节的

任务，之后，我们将会学习如何使用语法分析树来构建语言类应用程序。

2.3 你再也不能往核反应堆多加水了

歧义性语句是指存在不止一种语义的语句。换句话说，歧义性语句中的单词序列能够匹配多种语法结构。本节的标题“你再也不能往核反应堆多加水了”就是我在几年前的《周六夜现场》中看到的一个有歧义的句子。这句话让人不确定，是已经无法往核反应堆多加水了，还是不应该往核反应堆多加水。

我谢谢他了

我很喜欢的歧义句之一来源于我的朋友Kevin的博士文献词：“致我的博士生导师，我谢谢他了。”这句话让人不清楚他对他的博士导师的态度究竟是感激还是怨恨。Kevin本人声称是后者，所以我就问他，那为什么还要读这个导师的博士后。他回答道：“为了复仇。”

出现在自然语言中的歧义句会显得非常滑稽，但是出现在基于计算机的语言类应用程序中的歧义就会带来很多问题。为了解释或者翻译一个词组，程序必须能够唯一地辨识出它的准确含义。这意味着，我们必须提供没有歧义的语法，使得ANTLR生成的语法分析器能够以单一方式匹配每个输入词组。

迄今为止，我们还没有深入了解ANTLR语法的细节，不过，接下来我们将通过一些有歧义的语法来阐明歧义性的含义。你可以在以后构建语法并遇到歧义问题的时候再来回顾本节。

比如一些语法的歧义是非常明显的：

```
stat: ID '=' expr ';' // 匹配一个赋值语句⊖
    | ID '=' expr ';' // 糟糕！重复了前一个备选分支！
    ;
expr: INT ;
```

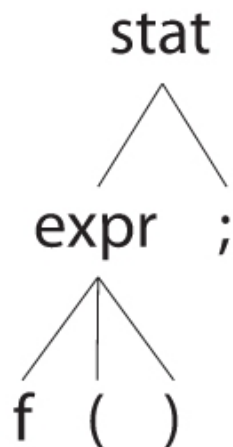
（注：原文为match an assignment; can match“f () ;”，我认为后半句位置有误，应移至下一段代码处。——译者注）

大多数情况下，歧义的表现更为微妙。在下面的语法中，**stat**规则包含两个备选分支，二者都可以匹配一个函数调用语句。

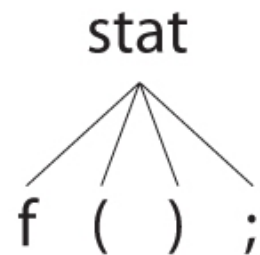
```
stat: expr ';' // 表达式语句，也可以匹配 "f();"
    | ID '(' ')' ';' // 函数调用语句
    ;
expr: ID '(' ')'
    | INT
    ;
```

下面的图显示了**stat**规则对输入文本“f () ;”的两种不同的解释：

f()); 作为表达式



f()); 作为函数调用



左边的语法分析树展示的是`f ()` 匹配`expr`规则的情况，右边的语法分析树展示的是`f ()` 匹配`stat`规则的第二个备选分支的情况。由于大多数语言的设计者都倾向于将语法设计成无歧义的，一个歧义性语法通常被认为是程序设计上的bug。我们需要重新组织语法，使得对于每个输入的词组，语法分析器都能够选择唯一匹配的备选分支。如果语法分析器检测到该词组存在歧义，它就必须多个备选分支中做出选择。ANTLR解决歧义问题的方法是：选择所有匹配的备选分支中的第一条。在上面的例子中，ANTLR将会选择左边的语法分析树作为对输入文本“`f () ;`”的语义解释。

歧义问题在词法分析器和语法分析器中都会发生，ANTLR的解决方案使得对规则的解析能够正常进行。在词法分析器中，ANTLR解决歧义问题的方法是：匹配在语法定义中最靠前的那条词法规则。我们通过

编程语言中常见的一种歧义——关键字和标识符规则的冲突——来说明这套机制是如何工作的。关键字**begin**同时也是一个标识符，至少从词法意义上来说是这样的。所以词法分析器可以使用以下任一词法规则来匹配字符序列“**b-e-g-i-n**”：

```
BEGIN : 'begin' ; // 匹配 b-e-g-i-n 序列，这存在歧义
ID    : [a-z]+ ;  // 匹配一个或者多个小写字母
```

有关词法分析中歧义性的更多信息，请参阅5.5节中“匹配标识符”部分。要注意的是，词法分析器会匹配可能的最长字符串来生成一个词法符号，这意味着，输入文本**beginner**只会匹配上例中的ID这条词法规则。ANTLR词法分析器不会把它匹配为关键字BEGIN后跟着标识符**ner**。

有时候，一门语言的语法本身就存在歧义，无论如何修改语法也不能改变这一点。例如，常见的数学表达式 $1+2*3$ 可以用两种方式解释，一种是自左向右地处理（**Smalltalk**就是这么做的），另外一种像是像绝大多数编程语言一样，按照优先级来处理。我们将在5.4节中学习如何隐式地指定表达式中的运算符优先级。

经典的C语言向我们展示了另外一种歧义，我们可以通过包含标识符定义的上下文信息来解决这样的歧义问题。例如，对于代码片段“**i*j**；”，从句法角度看，它像是一个表达式，但是实际上它的实际含义，或者说语义，依赖于**i**是一个类型还是一个变量。如果**i**是一个类

型的名字，那么这段代码就不是一个表达式，而是一个指向类型*i*的指针变量*j*的声明。我们将在第11章中解决这样的歧义问题。

语法分析器本身仅仅验证输入语句的合法性并建立一棵语法分析树。这是一项非常重要的工作，接下来，我们将了解一个语言类应用程序如何使用语法分析树来对输入文本进行语义分析和翻译。

2.4 使用语法分析树来构建语言类应用程序

为了编写一个语言类应用程序，我们必须对每个输入的词组或者子词组执行一些适当的操作。进行这项工作最简单的方式是操作语法分析器自动生成的语法分析树。这种方式的优点在于，我们能够重回我们所熟悉的Java领域。这样，在语言类应用程序进一步的构建过程中，我们就不需要再学习复杂的ANTLR语法了。

首先，我们来认识一下ANTLR在识别和建立语法分析树的过程中使用的数据结构和类名。熟悉这些数据结构将为我们未来的讨论奠定基础。

前已述及，词法分析器处理字符序列并将生成的词法符号提供给语法分析器，语法分析器随即根据这些信息来检查语法的正确性并建造出一棵语法分析树。这个过程对应的ANTLR类是CharStream、Lexer、Token、Parser，以及ParseTree。连接词法分析器和语法分析器的“管

道”就是`TokenStream`。图2-2展示了这些类型的对象在内存中的交互方式。

ANTLR尽可能多地使用共享数据结构来节约内存。如图2-2所示，语法分析树中的叶子节点（词法符号）仅仅是盛放词法符号流中的词法符号的容器。每个词法符号都记录了自己在字符序列中的开始位置和结束位置，而非保存子字符串的拷贝。其中，不存在空白字符对应的词法符号（索引为2和4的字符）的原因是，我们假定我们的词法分析器会丢弃空白字符。

图2-2中也显示出，`ParseTree`的子类`RuleNode`和`TerminalNode`，二者分别是子树的根节点和叶子节点。`RuleNode`有一些令人熟悉的方法，例如`getChild()`和`getParent()`，但是，对于一个特定的语法，`RuleNode`并不是确定不变的。为了更好地支持对特定节点的元素访问，ANTLR会为每条规则生成一个`RuleNode`的子类。如图2-3所示，在我们的赋值语句的例子中，子树根节点的类型实际上是`StatContext`、`AssignContext`以及`ExprContext`。

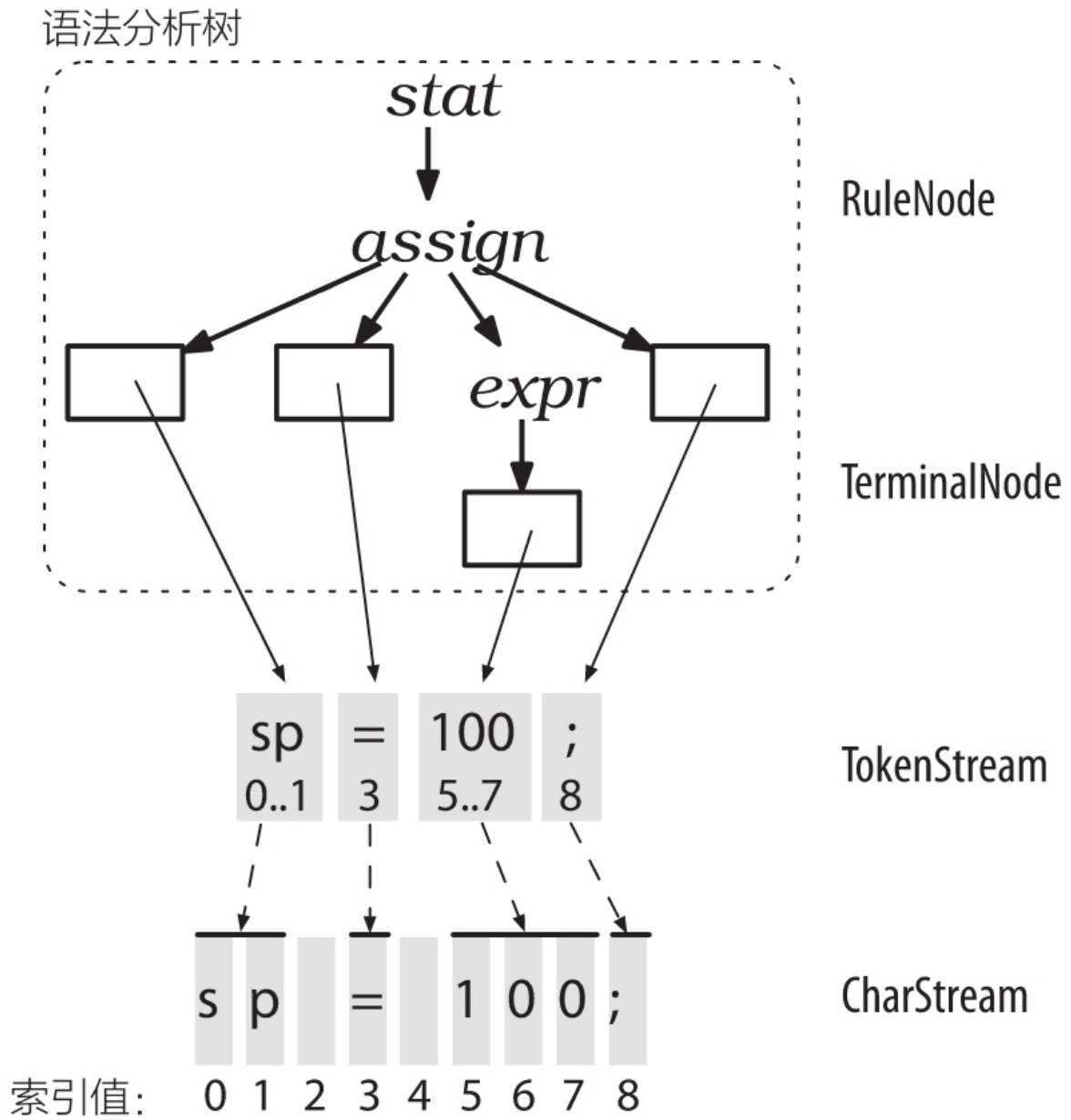
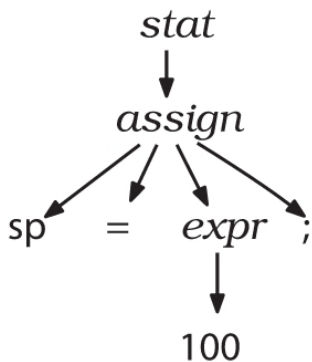
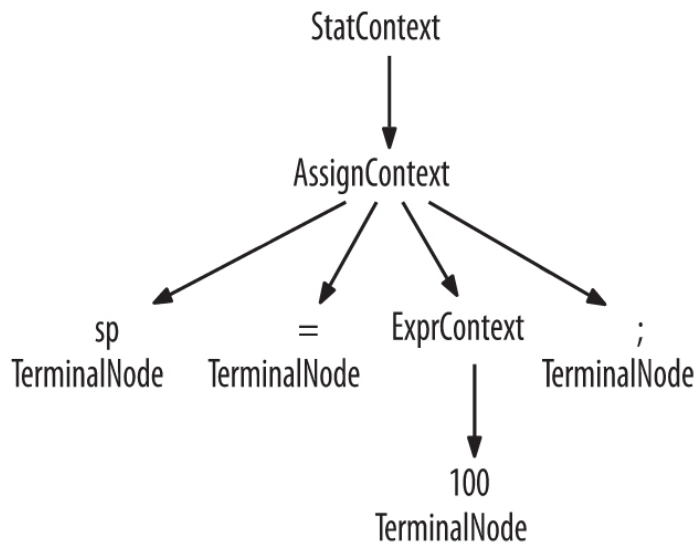


图2-2 部分对象在内存中的交互方式



a) 语法分析树



b) 语法分析树中各节点类名

图2-3 语法分析树

因为这些根节点包含了使用规则识别词组过程中的全部信息，它们被称为上下文（**context**）对象。每个上下文对象都知道自己识别出的词组中，开始和结束位置处的词法符号，同时提供访问该词组全部元素的途径。例如，**AssignContext**类提供了方法**ID（）**和方法**expr（）**来访问标识符节点和代表表达式的子树。

给定这些类型的具体实现，我们可以手工写出对语法分析树进行深度优先遍历的代码。这样，在访问其中的节点时，我们可以进行一切所需的操作。这个过程典型操作是诸如计算结果、更新数据结构或者产生输出一类的事情。实际上，我们可以利用**ANTLR**自动生成并遍历树的机制，而不需要每次都重复编写遍历树的代码。

2.5 语法分析树监听器和访问器

ANTLR的运行库提供了两种遍历树的机制。默认情况下，ANTLR使用内建的遍历器访问生成的语法分析树，并为每个遍历时可能触发的事件生成一个语法分析树监听器接口（`parse-tree listener interface`）。监听器非常类似于XML解析器生成的SAX文档对象。SAX监听器接收类似`startDocument()`和`endDocument()`的事件通知。一个监听器的方法实际上就是回调函数，正如我们在图形界面程序中响应复选框点击事件一样。除了监听器的方式，我们还将介绍另外一种遍历语法分析树的方式：访问者模式（`visitor pattern`）。

1. 语法分析树监听器

为了将遍历树时触发的事件转化为监听器的调用，ANTLR运行库提供了`ParseTreeWalker`类。我们可以自行实现`ParseTreeListener`接口，在其中填充自己的逻辑代码（通常是调用程序的其他部分），从而构建出我们自己的语言类应用程序。

ANTLR为每个语法文件生成一个`ParseTreeListener`的子类，在该类中，语法中的每条规则都有对应的`enter`方法和`exit`方法。例如，当遍历器访问到`assign`规则对应的节点时，它就会调用`enterAssign()`方法，然后将对应的语法分析树节点——`AssignContext`的实例——当作参数传递给它。在遍历器访问了`assign`节点的全部子节点之后，它会调用

`exitAssign()`。图2-4用粗虚线标识了ParseTreeWalker对语法分析树进行深度优先遍历的过程。

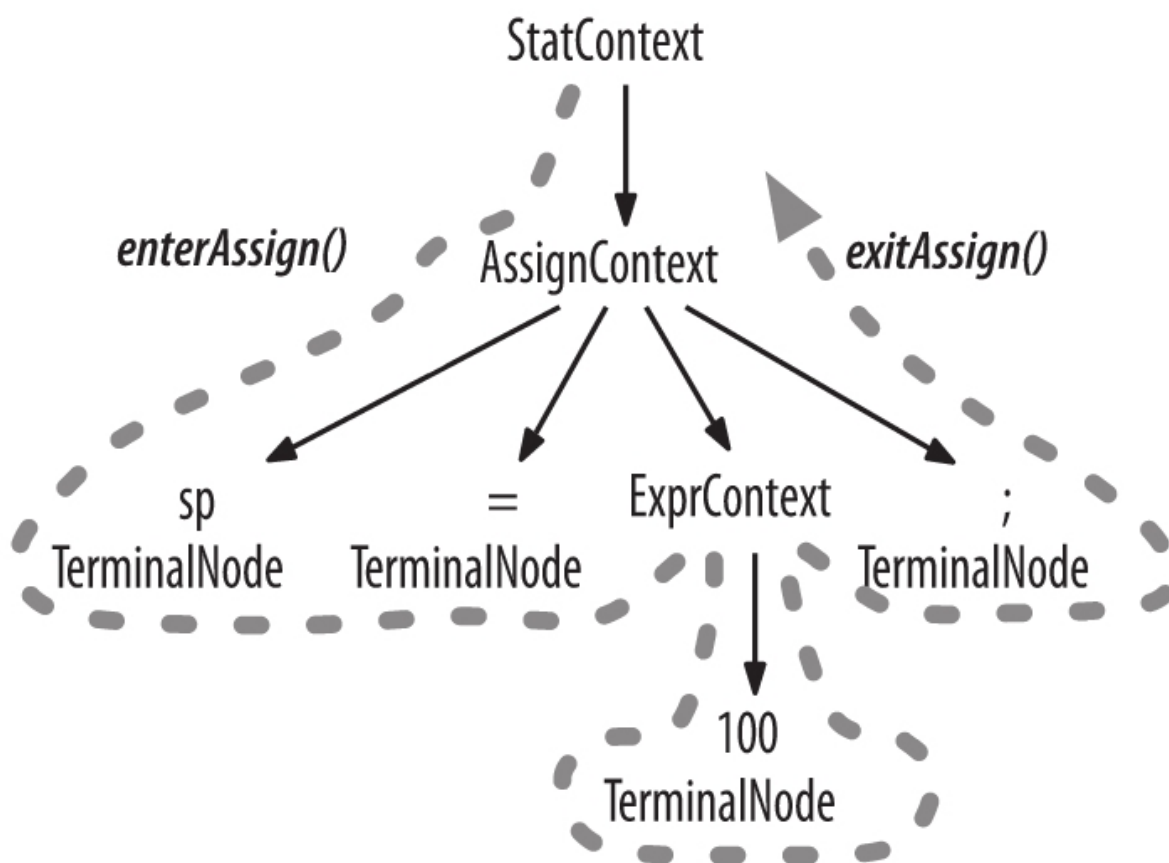


图2-4 ParseTreeWalker对语法分析树进行深度优先遍历的过程

除此之外，图2-4中还标识出了遍历过程中ParseTreeWalker调用assign规则的enter和exit方法的时机（其中未显示监听器其他方法的调用）。图2-5显示了在我们的赋值语句生成的语法分析树中，ParseTreeWalker对监听器方法的完整的调用顺序。

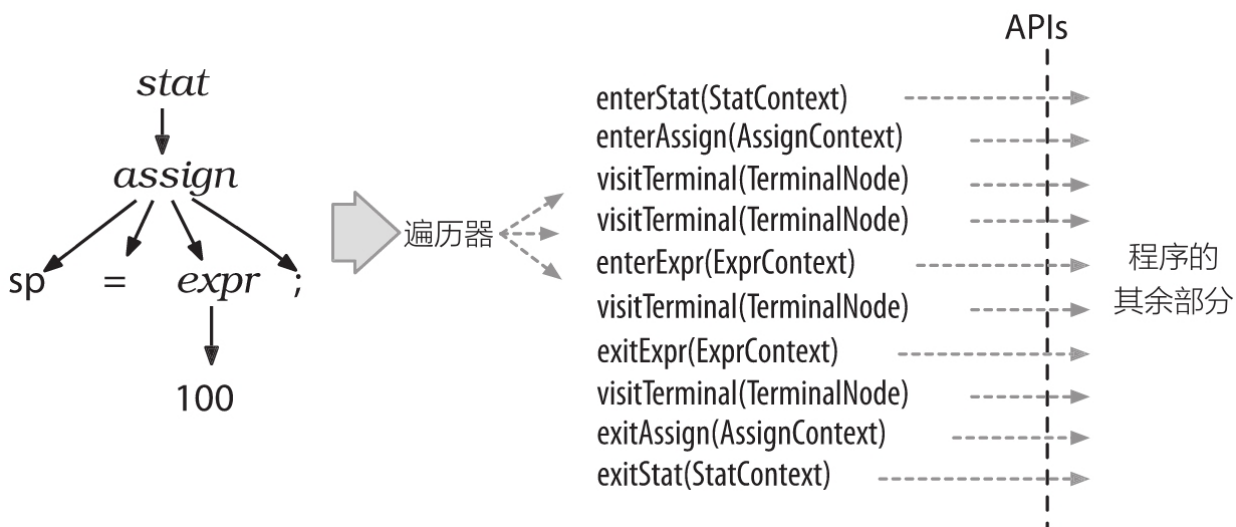


图2-5 ParseTreeWalker调用序列

监听器机制的优秀之处在于，这一切都是自动进行的。我们不需要编写对语法分析树的遍历代码，也不需要让我们的监听器显式地访问子节点。

2. 语法分析树访问器

有时候，我们希望控制遍历语法分析树的过程，通过显式的方法调用来访问子节点。在命令行中加入`-visitor`选项可以指示ANTLR为一个语法生成访问器接口（**visitor interface**），语法中的每条规则对应接口中的一个**visit**方法。图2-6是使用常见的访问者模式对我们的语法分析树进行操作的过程。

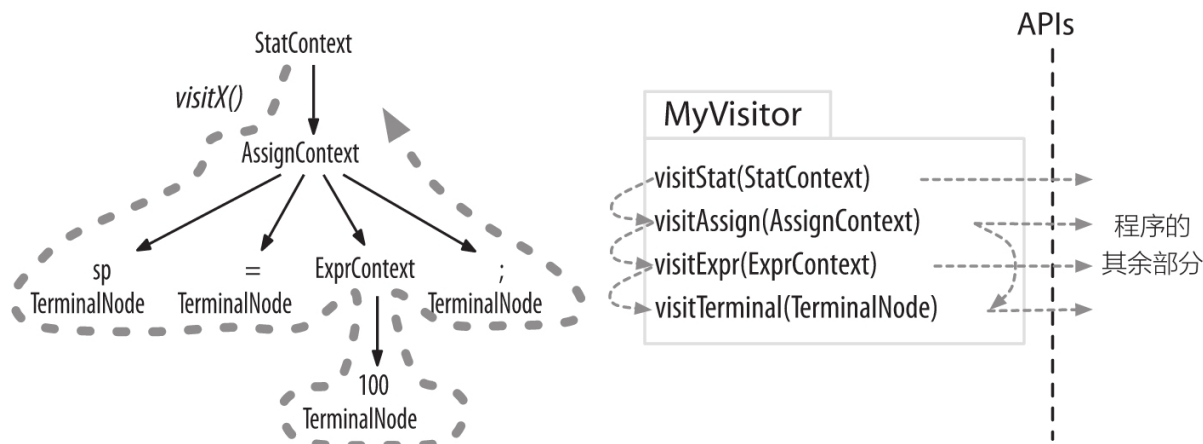


图2-6 使用常见的访问者模式对语法分析树进行操作的过程

其中，粗虚线显示了对语法分析树进行深度优先遍历的过程。细虚线标示出访问器方法的调用顺序。我们可以在自己的程序代码中实现这个访问器接口，然后调用**visit**（）方法来开始对语法分析树的一次遍历。

```
ParseTree tree = ... ; // tree 是语法分析得到的结果
MyVisitor v = new MyVisitor();
v.visit(tree);
```

ANTLR内部为访问者模式提供的支持代码会在根节点处调用**visitStat**（）方法。接下来，**visitStat**（）方法的实现将会调用**visit**（）方法，并将所有子节点当作参数传递给它，从而继续遍历的过程。或者，**visitMethod**（）方法可以显式调用**visitAssign**（）方法等。

ANTLR会提供访问器接口和一个默认实现类，免去我们一切都要自行实现的麻烦。这样，我们就可以专注于那些我们感兴趣的方法，而无

须覆盖接口中的方法。我们将在第7章中深入介绍访问器和监听器。

与语法分析相关的术语

本章介绍了很多重要的与语言识别相关的术语。

语言 一门语言是一个有效语句的集合。语句由词组组成，词组由子词组组成，子词组又由更小的子词组组成，依此类推。

语法 语法定义了语言的语义规则。语法中的每条规则定义了一种词组结构。

语义树或语法分析树 代表了语句的结构，其中的每个子树的根节点都使用一个抽象的名字给其包含的元素命名。即子树的根节点对应了语法规则的名字。树的叶子节点是语句中的符号或者词法符号。

词法符号 词法符号就是一门语言的基本词汇符号，它们可以代表像是“标识符”这样的一类符号，也可以代表一个单一的运算符，或者代表一个关键字。

词法分析器或者词法符号生成器 将输入的字符序列分解成一系列词法符号。一个词法分析器负责分析词法。

语法分析器 语法分析器通过检查语句的结构是否符合语法规则的定义来验证该语句在特定语言中是否合法。语法分析的过程好比是走迷

宫，通过比较语句中和地板上的单词来从入口走到出口。ANTLR能够生成被称为ALL（*）的自顶向下的语法分析器，ALL（*）是指它可以利用剩余的所有输入文本来进行决策。自顶向下的语法分析器以结果为导向，首先匹配最粗粒度的规则，这样的规则通常命名为program或者inputFile。

递归下降的语法分析器 这是自顶向下的语法分析器的一种实现，每条规则都对应语法分析器中的一个函数。

前向预测 语法分析器使用前向预测来进行决策，具体方法是：将输入的符号与每个备选分支的起始符号进行比较。

迄今为止，我们已经大体上了解了ANTLR的工作原理。在本章中，我们认识了从字符序列到语法分析树的整个流程，学习了ANTLR运行库的一些关键类。此外，我们还简单了解了监听器和访问器机制，它们是连接语法分析器和特定程序代码的桥梁。在下一章中，我们将通过一个实际的例子来使大家加深对上述概念的理解。

第3章 入门的ANTLR项目

作为我们的第一个ANTLR项目，我们会构造一个语法，它是C语言或其继承者Java语法的一个很小的子集。具体来说，我们将识别包裹在花括号或者嵌套的花括号中的一些整数，像是{1, 2, 3}和{1, {2, 3}},

4}这样。这样的结构可以作为int数组或者C语言中的结构体的初始化语句。在很多情况下，针对这种语法的语法分析器都非常有用。例如，我们可以用它来构建一个对C语言的源代码进行重构的工具，这个工具能够完成这样的工作：如果初始化语句中所有的整数值都能用一个字节表示，那么将该整数数组转换为字节数组。我们也可以用这个语法分析器将Java的short数组转换为字符串。例如，我们可以将short值当作Unicode字符，从而将：

```
static short[] data = {1,2,3};
```

转换为等价的字符串形式：

```
static String data = "\u0001\u0002\u0003"; // Java 中的 char 实际上是 unsigned short
```

其中像\u0001这样的Unicode字符标记使用四个十六进制数字来表示一个16位的字符。实际上，这样的字符就是一个short值。

我们这样做的原因是为了不受Java的.class文件格式的限制。Java的.class文件将数组的初始化语句存储为一系列显式的数组元素赋值语句，上面的初始化语句等价于data[0]=1; data[1]=2; data[2]=3;。这限制了我們能够使用这种方法来初始化的数组的大小。相比之下，Java的.class文件将字符串存储为连续的short序列，从而不受上述约束限制。将数组的初始化语句转换为字符串可以得到更紧凑的.class文件，避免了Java的对初始化方法的长度限制。

通过这个入门的项目示例，你将会学到如下内容：一些ANTLR语法的语义元素定义、ANTLR根据语法自动生成代码的机制、如何将自动生成的语法分析器和Java程序集成，以及如何使用语法分析树监听器编写一个代码翻译工具。

3.1 ANTLR工具、运行库以及自动生成的代码

在开始前，我们先浏览一下ANTLR的jar包中的内容。在ANTLR的jar包中存在两个关键部分：ANTLR工具和ANTLR运行库（运行时语法分析）API。通常，当说到“对一个语法运行ANTLR”时，我们指的是运行ANTLR工具，即`org.antlr.v4.Tool`类来生成一些代码（语法分析器和词法分析器），它们能够识别使用这份语法代表的语言所写成的语句。词法分析器将输入的字符流分解为词法符号序列，然后将它们传递给能够进行语法检查的语法分析器。运行库是一个由若干类和方法组成的库，这些类和方法是自动生成的代码（如Parser，Lexer和Token）运行所必须的。因此，我们完成工作的一般步骤是：首先我们对一个语法运行ANTLR，然后将生成的代码与jar包中的运行库一起编译，最后将编译好的代码和运行库放在一起运行。

构建一个语言类应用程序的第一步是创建一个能够描述这种语言的语法（即合法语句结构的集合）的语法。我们将在第5章中介绍如何编写语法，现在我们先来看下面这个能够满足我们需求的语法。

starter/ArrayInit.g4

```
/** 语法文件通常以 grammar 关键字开头
 * 这是一个名为 ArrayInit 的语法，它必须和文件名 ArrayInit.g4 相匹配
 */
grammar ArrayInit;

/** 一条名为 init 的规则，它匹配一对花括号中的、逗号分隔的 value */
init : '{' value (',' value)* '}' ; // 必须匹配至少一个 value

/** 一个 value 可以是嵌套的花括号结构，也可以是一个简单的整数，即 INT 词法符号 */
value : init
      | INT
      ;

// 语法分析器的规则必须以小写字母开头，词法分析器的规则必须用大写字母开头
INT : [0-9]+ ; // 定义词法符号 INT，它由一个或多个数字组成
WS : [\t\r\n]+ -> skip ; // 定义词法规则“空白符号”，丢弃之
```

请将语法文件ArrayInit.g4放入一个单独的文件夹，例如/tmp/array（通过复制-粘贴或者从本书网站下载）。然后我们对它运行ANTLR工具。

```
$ cd /tmp/array
$ antlr4 ArrayInit.g4 # 使用 antlr4 这个别名命令来生成语法分析器和词法分析器
```

根据语法ArrayInit.g4，ANTLR自动生成了很多文件，如图3-1所示，正常情况下这些文件都是需要我们手工编写的。

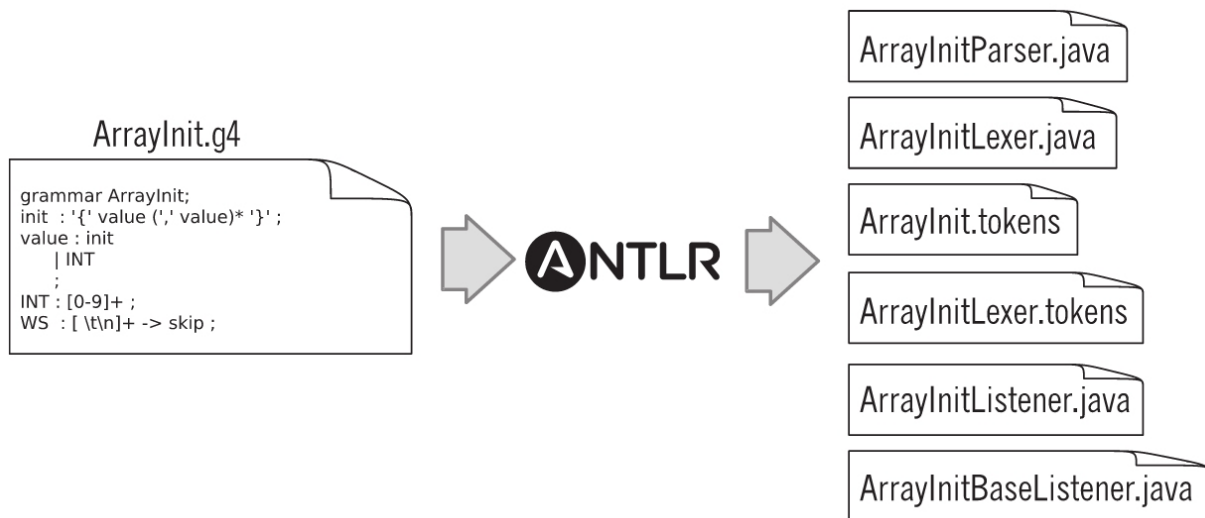


图3-1 根据语法ArrayInit.g4，ANTLR生成的文件

目前，我们仅仅需要大致了解这个过程，下面简单介绍一下生成的文件：

1) **ArrayInitParser.java**: 该文件包含一个语法分析器类的定义，这个语法分析器专门用来识别我们的“数组语言”的语法**ArrayInit**。

```
public class ArrayInitParser extends Parser { ... }
```

在该类中，每条规则都有对应的方法，除此之外，还有一些其他的辅助代码。

2) **ArrayInitLexer.java**: ANTLR能够自动识别出我们的语法中的文法规则和词法规则。这个文件包含的是词法分析器的类定义，它是由ANTLR通过分析词法规则**INT**和**WS**，以及语法中的字面值{'、', '，

和'}'生成的。回想一下上一章的内容，词法分析器的作用是将输入字符序列分解成词汇符号。它形如：

```
public class ArrayInitLexer extends Lexer { ... }
```

3) **ArrayInit.tokens**: ANTLR会给每个我们定义的词法符号指定一个数字形式的类型，然后将它们的对应关系存储于该文件中。有时，我们需要将一个大型语法切分为多个更小的语法，在这种情况下，这个文件就非常有用。通过它，ANTLR可以在多个小型语法间同步全部的词法符号类型。更多内容请参阅4.1节中的“语法导入”部分。

4) **ArrayInitListener.java**, **ArrayInitBaseListener.java**: 默认情况下，ANTLR生成的语法分析器能将输入文本转换为一棵语法分析树。在遍历语法分析树时，遍历器能够触发一系列“事件”（回调），并通知我们提供的监听器对象。**ArrayInitListener**接口给出了这些回调方法的定义，我们可以实现它来完成自定义的功能。**ArrayInitBaseListener**是该接口的默认实现类，为其中的每个方法提供了一个空实现。

ArrayInitBaseListener类使得我们只需要覆盖那些我们感兴趣的回调方法（详见7.2节）。通过指定**-visitor**命令行参数，ANTLR也可以为我们生成语法分析树的访问器（参阅7.5节“使用访问器遍历语法分析树”部分）。

接下来，我们将使用监听器来将short数组初始化语句转换为字符串对象，不过在这之前，我们首先使用一些样例输入来验证我们的语法分析器是否能正常进行匹配工作。

ANTLR语法比正则表达式功能更强大

熟悉正则表达式的读者可能会产生疑问，使用ANTLR来解决这么简单的识别问题是不是有点小题大做了？实际上，由于嵌套的花括号结构的存在，正则表达式无法识别这样的初始化语句。正则表达式没有存储的概念，它们无法记住之前匹配过的输入。因此，它们不能将左右花括号正确配对。我们将在5.3节“嵌套模式”部分中予以详细讨论。

3.2 测试生成的语法分析器

对语法运行ANTLR之后，我们需要编译自动生成的Java源代码。简单起见，我们在工作目录/tmp/array下完成所有的编译操作。

```
$ cd /tmp/array
$ javac *.java      # 编译 ANTLR 自动生成的代码
```

如果编译器产生了一个ClassNotFoundException异常，说明你可能没有正确设置Java的CLASSPATH环境变量。在类UNIX系统上，你需要执行以下命令（也可以写在类似.bash_profile的启动脚本中）：

```
$ export CLASSPATH=".:usr/local/lib/antlr-4.0-complete.jar:$CLASSPATH"
```


我们使用第1章提到的`grun`别名来启动**TestRig**，执行对语法的测试。下面的命令告诉我们如何将词法分析器生成的词法符号打印出来。

```
⇒ $ grun ArrayInit init -tokens
⇒ {99, 3, 451}
⇒ EOF
  < [@0,0:0='{',<1>,1:0]
    [@1,1:2='99',<4>,1:1]
    [@2,3:3=',',<2>,1:3]
    [@3,5:5='3',<4>,1:5]
    [@4,6:6=',',<2>,1:6]
    [@5,8:10='451',<4>,1:8]
    [@6,11:11='}',<3>,1:11]
    [@7,13:12='<EOF>',<-1>,2:0]
```

在输入要测试的语句{99, 3, 451}之后，我们必须手动输入一个EOF。默认情况下，**ANTLR**在开始处理前会加载全部的输入文本（这是最常见的情况，也是最有效率的处理方式）。

每行输出代表一个词法符号，其中包含该词法符号的全部信息。例如，`[@5, 8: 10='451', <4>, 1: 8]`表明它是第5个词法符号（从0开始计数），由第8到第10个字符组成（从0开始计数，包含第8和第10），包含的文本是451，类型是4（INT），位于输入文本的第1行（从1开始计数）第8个字符（从0开始计数，`tab`作为一个字符）处。注意，输出的结果中不包含空格和换行符，这是因为在我们的语法中，**WS**规则的“`->skip`”指令将它们丢弃了。

如果需要语法分析器关于输入文本识别过程的更多信息，我们可以使用“`-tree`”选项查看语法分析树：

```
⇒ $ grun ArrayInit init -tree
⇒ {99, 3, 451}
⇒ EOF
  < (init { (value 99) , (value 3) , (value 451) })
```

“-tree”选项使用LISP风格（根节点和子节点在同一行显示）打印出语法分析树。另外，我们也可以使用“-gui”选项生成一个可视化的对话框。我们试着用这个选项处理一下{1, {2, 3}, 4}这样的嵌套结构。

```
⇒ $ grun ArrayInit init -gui
⇒ {1,{2,3},4}
⇒ EOF
```

图3-2就是弹出的包含语法分析树的对话框。

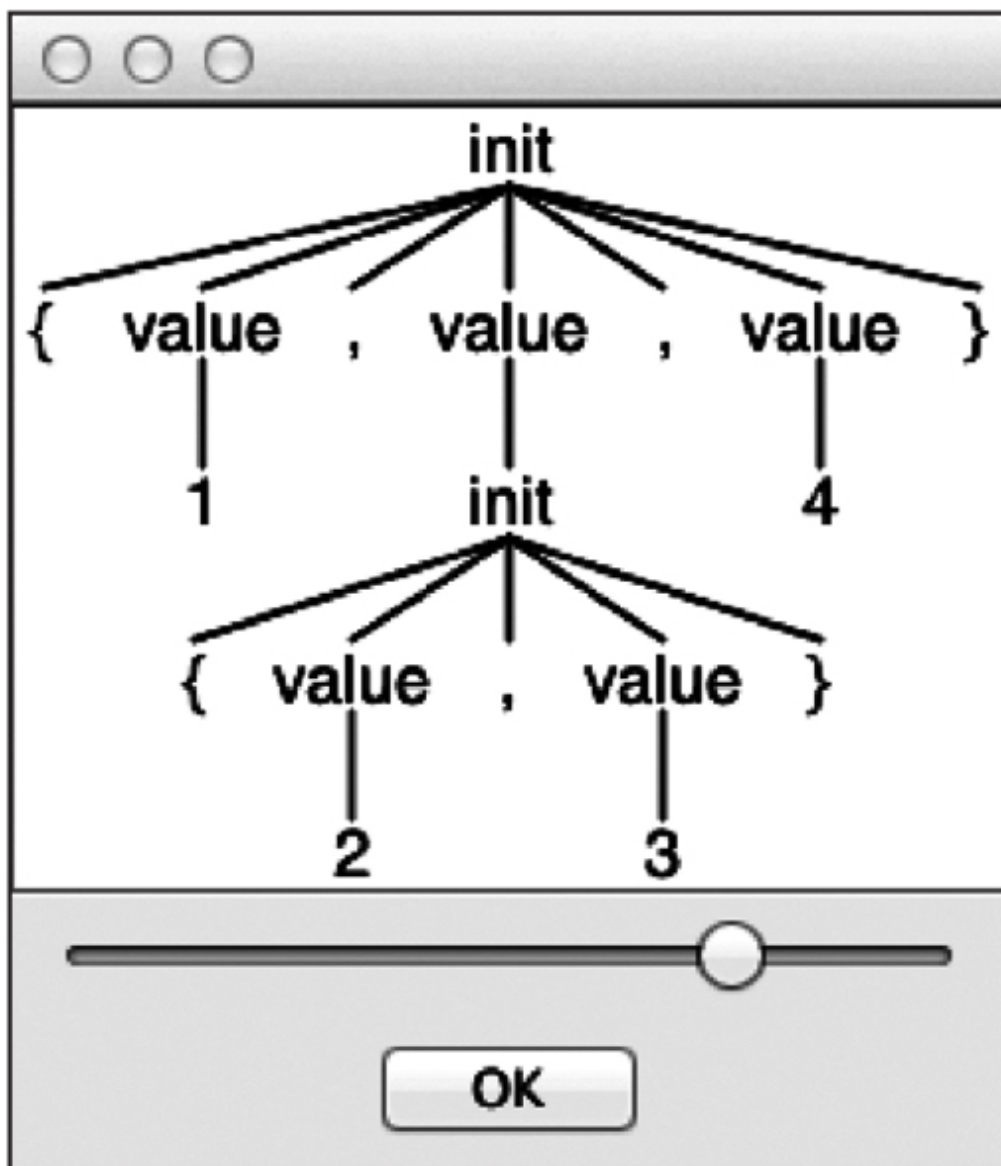


图3-2 包含语法分析树的示例对话框

用自然语言表述，语法分析树就是，“输入的是一个由一对花括号包裹的三个值组成的初始化语句，第一个和第三个值是整数1和4，第二个值也是一个初始化语句，它由一对花括号包裹的两个值组成，这两个值是整数2和3”。

这些内部节点，即init节点和value节点，非常通俗易懂，因为它们用名字标识出了复杂的输入元素。这有点像是标识英语句子中的动词和主语。ANTLR最棒的部分在于，它能够基于我们的语法中的规则名自动创建这样的一棵语法分析树。在本章的最后，我们会使用ANTLR内置的语法分析树遍历器触发自定义的回调函数enterInit () 和enterValue ()，从而构建出一个满足要求的翻译器。

现在我们已经能用ANTLR分析语法、生成代码，并且测试它们了，接下来我们要思考的是，如何从另外一个Java程序中调用生成的语法分析器。

3.3 将生成的语法分析器与Java程序集成

在语法准备就绪之后，我们就可以将ANTLR自动生成的代码和一个更大的程序进行集成。在本节中，我们将会使用一个简单的Java示例程序的main () 方法调用我们的“初始化语句解析器”，并打印出和TestRig的“-tree”选项类似的语法分析树。下面是完整的Test.java程序，它体现出了2.1节中的完整的识别流程。

starter/Test.java

```
// 导入 ANTLR 的运行库
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class Test {
    public static void main(String[] args) throws Exception {
        // 新建一个 CharStream, 从标准输入读取数据
        ANTLRInputStream input = new ANTLRInputStream(System.in);

        // 新建一个词法分析器, 处理输入的 CharStream
        ArrayInitLexer lexer = new ArrayInitLexer(input);

        // 新建一个词法符号的缓冲区, 用于存储词法分析器将生成的词法符号
        CommonTokenStream tokens = new CommonTokenStream(lexer);

        // 新建一个语法分析器, 处理词法符号缓冲区中的内容
        ArrayInitParser parser = new ArrayInitParser(tokens);

        ParseTree tree = parser.init(); // 针对 init 规则, 开始语法分析
        System.out.println(tree.toStringTree(parser)); // 用 LISP 风格打印生成的树
    }
}
```

上面的程序使用了很多ANTLR运行库的类, 像是CommonTokenStream和ParseTree, 我们将在4.1节中深入学习它们。

下面是编译运行Test的方式:

```
⇒ $ javac ArrayInit*.java Test.java
⇒ $ java Test
⇒ {1,{2,3},4}
⇒ EoF
⚡ (init { (value 1) , (value (init { (value 2) , (value 3) })) , (value 4) })
```

ANTLR还能自动报告语法错误, 并从语法错误中恢复。例如, 如果我们输入一个缺失最后的右花括号的初始化语句, 结果会是下面这样:

```
⇒ $ java Test
⇒ {1,2
⇒ EOF
< line 2:0 missing '}' at '<EOF>'
  (init { (value 1) , (value 2) <missing '}'>)
```

现在，我们已经知道了如何对一个语法运行ANTLR工具，以及如何将自动生成的语法分析器和一个微型的Java程序集成。不过，一个仅仅能够检查语法正确性的程序并没有什么亮点，我们要构建的是一个能够将short数组初始化语句转换为String对象的翻译器。

3.4 构建一个语言类应用程序

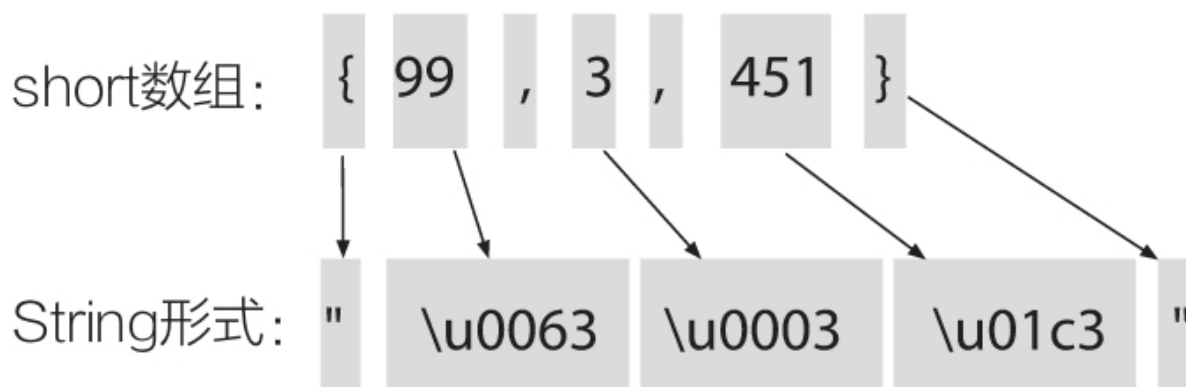
我们继续完成能够处理数组初始化语句的示例程序，下一个目标是能够翻译初始化语句，而不仅仅是能够识别它们。例如，我们想要将Java中，类似{99, 3, 451}的short数组翻译成"0063\u0003\u01c3"。注意，其中十进制数字99的十六进制表示是63。

为了完成这项工作，程序必须能够从语法分析树中提取数据。最简单的方案是使用ANTLR内置的语法分析树遍历器进行深度优先遍历，然后在它触发的一系列回调函数中进行适当的操作。正如我们之前看到的那样，ANTLR能够自动生成一个监听器接口和一个默认的实现类。这样的监听器非常类似于图形界面程序控件上的回调函数（例如，当一个按钮被按下时，它会通知我们）或者XML解析器中的SAX事件。

我们如果希望通过编写程序来操纵输入的数据的话，只需要继承 `ArrayInitBaseListener` 类，然后覆盖其中必要的方法即可。我们的基本思想是，在遍历器进行语法分析树的遍历时，令每个监听器方法翻译输入数据的一部分并将结果打印出来。

监听器机制的优雅之处在于，我们不需要自己编写任何遍历语法分析树的代码。事实上，我们甚至都不知道 `ANTLR` 运行库是怎么遍历语法分析树、怎么调用我们的方法的。我们只知道，在语法规则对应的语句的开始和结束位置处，我们的监听器方法可以得到通知。在 7.2 节我们会看到，这种机制使得我们不需要了解太多 `ANTLR` 的知识——我们回到了自己熟悉的领域，即写普通的代码而不是处理语言识别问题。

一个进行翻译工作的项目意味着要处理这样的问题：如何将输入的词法符号或者词组翻译成输出字符串。为了达到这个目标，最好先从手工翻译一些有代表性的样例入手，想办法提取出通用的转换逻辑。在下例中，转换过程是非常直截了当的。



用自然语言解释，翻译过程就是一系列“X映射为Y”的过程：

- 1) 将{翻译为"。
- 2) 将}翻译为"。
- 3) 将每个整数翻译为四位的十六进制形式，然后加前缀。

为此，我们需要编写方法，在遇到对应的输入词法符号或者词组的时候，打印出转换后的字符串。内置的语法分析树遍历器会在各种词组的开始和结束位置触发监听器的回调函数。下面是遵循我们的翻译规则的一个监听器的实现类。

starter/ShortToUnicodeString.java

```
/** 将类似 {1,2,3} 的 short 数组初始化语句翻译为 "\u0001\u0002\u0003" */
public class ShortToUnicodeString extends ArrayInitBaseListener {
    /** 将 { 翻译为 " */
    @Override
    public void enterInit(ArrayInitParser.InitContext ctx) {
        System.out.print('');
    }

    /** 将 } 翻译为 " */
    @Override
    public void exitInit(ArrayInitParser.InitContext ctx) {
        System.out.print('');
    }

    /** 将每个整数翻译为四位的十六进制形式，然后加前缀 \u */
    @Override
    public void enterValue(ArrayInitParser.ValueContext ctx) {
        // 假定不存在嵌套结构
        int value = Integer.valueOf(ctx.INT().getText());
        System.out.printf("\\u%04x", value);
    }
}
```


我们不需要覆盖每个enter/exit方法，我们只需要覆盖自己需要的那些。上述代码中唯一令我们不那么熟悉的一个表达式是ctx.INT（），它从上下文对象中获取INT词法符号对应的整数值，该词法符号由匹配value规则得来。现在，剩下的事情就是把之前的Test类扩展成一个翻译程序了。

```
starter/Translate.java
// 导入 ANTLR 运行库
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class Translate {
    public static void main(String[] args) throws Exception {
        // 新建一个 CharStream，从标准输入读取数据
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        // 新建一个词法分析器，处理输入的 CharStream
        ArrayInitLexer lexer = new ArrayInitLexer(input);
        // 新建一个词法符号的缓冲区，用于存储词法分析器将生成的词法符号
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        // 新建一个语法分析器，处理词法符号缓冲区中的内容
        ArrayInitParser parser = new ArrayInitParser(tokens);
        ParseTree tree = parser.init(); // 针对 init 规则，开始语法分析

        // 新建一个通用的、能够触发回调函数的语法分析树遍历器
        ParseTreeWalker walker = new ParseTreeWalker();

        // 遍历语法分析过程中生成的语法分析树，触发回调
        walker.walk(new ShortToUnicodeString(), tree);
        System.out.println(); // 翻译完成后，打印一个 \n
    }
}
```

这份代码和之前代码的唯一区别在于高亮标识的部分，它新建了一个语法分析树遍历器，令其对语法分析器生成的语法分析树进行遍历。在遍历器的遍历过程中，它触发了我们的ShortToUnicodeString监听器的回调函数。请注意：限于篇幅，为了使读者的注意力更加集中，在

本书剩下的章节中，通常仅给出代码的关键部分而非完整代码。此外，你还可以从本书的网站上获取完整的代码压缩包。

最后，让我们一起完成这个翻译器，并使用样例输入来测试。

```
⇒ $ javac ArrayInit*.java Translate.java
⇒ $ java Translate
⇒ {99, 3, 451}
⇒ EOf
  < "\u0063\u0003\u01c3"
```

一切顺利。无须深入理解语法的细节，我们就完成了我们的第一个翻译器。我们所做的一切不过是实现了几个方法，在这些方法中打印出对输入文本的适当的翻译结果。另外，我们可以通过给遍历器传递一个不同的监听器以实现完全不同的输出。监听器有效地将语言类应用程序和语法进行了解耦，从而使得同一个语法能够被不同的程序复用。

下一章，我们将快速学习ANTLR语法的编写方法，以及让ANTLR语法如此强大和易用的关键特性。

第4章 快速指南

迄今为止，我们已经学习了如何安装ANTLR，也知道了构建语言类应用程序所需的步骤、术语和原材料。本章中，我们将会通过几个功能强大的示例程序来快速上手ANTLR。在这个过程中，我们可能会省略

掉一些细节，所以如果你不明所以的话，也不必担心，我们的目标只是让你知道，使用**ANTLR**可以做什么。具体细节我们将在第5章中详细讲解。对于那些有过**ANTLR**早期版本使用经验的开发者来说，本章是一个很好的升级指南。

本章分为四个主题，分别展示了**ANTLR**的不同特性。在阅读本章时，最好下载本书的样例代码同步学习。这样，你就能习惯于编写语法文件和构建**ANTLR**程序。记住，遍布本章的代码片段并不是完整的文件，我们侧重代码的关键部分。

在第一个主题中，我们会分析一个简单的算术表达式语言的语法。我们会使用**ANTLR**内置的测试工具对它进行测试，然后深入学习在3.3节样例代码中所介绍的语法分析器的启动过程。之后，我们将会研究表达式语法生成的包含异常节点的语法分析树。（回想一下，语法分析树记录了语法分析器在匹配输入词组过程中的完整信息。）对于非常庞大的语法，我们将会学习如何使用语法导入（**grammar import**）将其切分为更容易管理的小块。最后，我们将了解**ANTLR**自动生成的语法分析器处理非法输入的机制。

在算术表达式语法分析器之后，进入第二个主题，我们将会使用访问者模式构建一个计算器，它能够遍历算术表达式的语法分析树并计算结果。**ANTLR**语法分析器能够自动生成访问器接口和空的实现类，让我们上手更加容易。

在第三个主题中，我们将会构建一个翻译器，它能够读取Java类定义并把它翻译成一个仅有方法声明的接口。我们通过同样由ANTLR自动生成的监听器机制完成这项工作。

在第四个主题中，我们将会学习如何将动作（**action**，即任意代码）直接嵌入语法文件。大多数情况下，我们都是通过访问器和监听器来构建语言类应用程序的，但是为了实现极端的灵活性，**ANTLR**允许直接将自定义的程序代码嵌入生成的语法分析器。这些动作是任意代码，在语法分析过程中执行，能够完成收集信息或者生成输出之类的事情。通过和语义判定（**semantic predicate**，即内嵌的布尔表达式）协同工作，我们甚至可以令部分语法在运行时消失！例如，在Java语法中，处理不同版本的源代码时，我们可能需要开启或者关闭**enum**关键字。如果没有语义判定功能，我们就必须编写两个版本的语法。

在最后一个主题中，我们将目光聚焦于词法分析（词法符号）层面上的一些**ANTLR**特性。我们将看到**ANTLR**是如何处理包含不止一种语言的输入文件的。之后，我们将了解一个很棒的类**TokenStreamRewriter**，它允许我们修改词法符号流而不影响原先的输入流。最后，我们将会回顾第一个主题中接口生成器的例子，学习**ANTLR**如何在解析Java语言时忽略输入的空白字符和注释，但是并不丢弃它们，而是留待以后处理。

现在开启ANTLR之旅，首先，我们要了解组成ANTLR语法的基本标记。请确保你的命令行已经为`antlr4`和`grun`设定了正确的别名，详见1.2节。

4.1 匹配算术表达式的语言

我们的第一个语法用于构建一个简单的计算器，其对算术表达式的处理具有十分重要的意义，因为它们太常见了。为简单起见，我们只允许基本的算术操作符（加减乘除）、圆括号、整数以及变量出现。我们的算术表达式限制浮点数的使用，只允许整数出现。

下面的示例包含了本语言的全部特性：

```
tour/t.expr
```

```
193
a = 5
b = 6
a+b*2
(1+2)*3
```

用自然语言来说，我们的表达式语言组成的程序就是一系列语句，每个语句都由换行符终止。一个语句可以是一个表达式、一个赋值语句或者是一个空行。下面是解析这样的赋值语句和表达式的ANTLR语法：

```
tour/Expr.g4
```

```
第 1 行 grammar Expr;
```

```
-
- /** 起始规则，语法分析的起点 */
- prog:  stat+ ;
```

```

5
- stat:  expr NEWLINE
-      |  ID '=' expr NEWLINE
-      |  NEWLINE
-      ;
10
- expr:  expr ('*' | '/') expr
-      |  expr ('+' | '-' ) expr
-      |  INT
-      |  ID
15      |  '(' expr ')'
-      ;
-
- ID   :   [a-zA-Z]+ ;           // 匹配标识符
- INT  :   [0-9]+ ;             // 匹配整数
20 NEWLINE: '\r'? '\n' ;        // 告诉语法分析器一个新行的开始（即语句终止标志）
- WS   :   [ \t]+ -> skip ;    // 丢弃空白字符

```

不过多地深究细节，让我们看一看ANTLR语法基本标记包含哪些元素。

- 语法包含一系列描述语言结构的规则。这些规则既包括类似stat和expr的描述语法结构的规则，也包括描述标识符和整数之类的词汇符号（词法符号）的规则。
- 语法分析器的规则以小写字母开头。
- 词法分析器的规则以大写字母开头。
- 我们使用|来分隔同一个语言规则的若干备选分支，使用圆括号把一些符号组合成子规则。例如，子规则（'*'|'/'）匹配一个乘法符号或者一个除法符号。

我们将在第5章中对此进行详细讨论。

ANTLR 4的最重要的新功能之一就是，它能够处理（大部分情况下的）左递归规则。左递归规则是指这样的语言规则：在某个备选分支的起始位置调用了自身。例如，在上述语法中，`expr`规则的第11行和第12行的备选分支在左侧递归调用了`expr`规则。使用这种方式指定算术表达式远比传统的自顶向下的语法分析器策略简单。在传统的语法分析器策略中，我们需要为运算符的每种优先级编写一条规则。有关左递归消除功能的更多信息，请参阅5.4节。

词法符号定义中的标记和正则表达式的元字符非常相似。我们将在第6章中介绍更多词法（词法符号）规则。现在，唯一不寻常的地方在于WS词法规则后面的“`->skip`”操作。它是一条指令，告诉词法分析器匹配并丢弃空白字符（每个输入的字符都必须被至少一条词法规则匹配）。通过使用正式的ANTLR标记，而非嵌入一段代码来告诉词法分析器忽略这些字符，我们就能避免语法和某种特定的目标语言绑定。

现在我们可以和Expr语法一起开始一段愉快的旅程了。复制/粘贴上面的tour/Expr.g4到适当的位置。测试语法最简单的方式是使用内置的TestRig，我们可以通过grun别名来使用它。例如，下面是在类UNIX系统的命令行中的构建和测试过程：

```
$ antlr4 Expr.g4
$ ls Expr*.java
```

```

ExprBaseListener.java    ExprListener.java
ExprLexer.java           ExprParser.java
$ javac Expr*.java
$ grun Expr prog -gui t.expr # 启动 org.antlr.v4.runtime.misc.TestRig

```

由于我们添加了“-gui”选项，测试组件弹出了一个展示语法分析树的窗口，如图4-1所示。

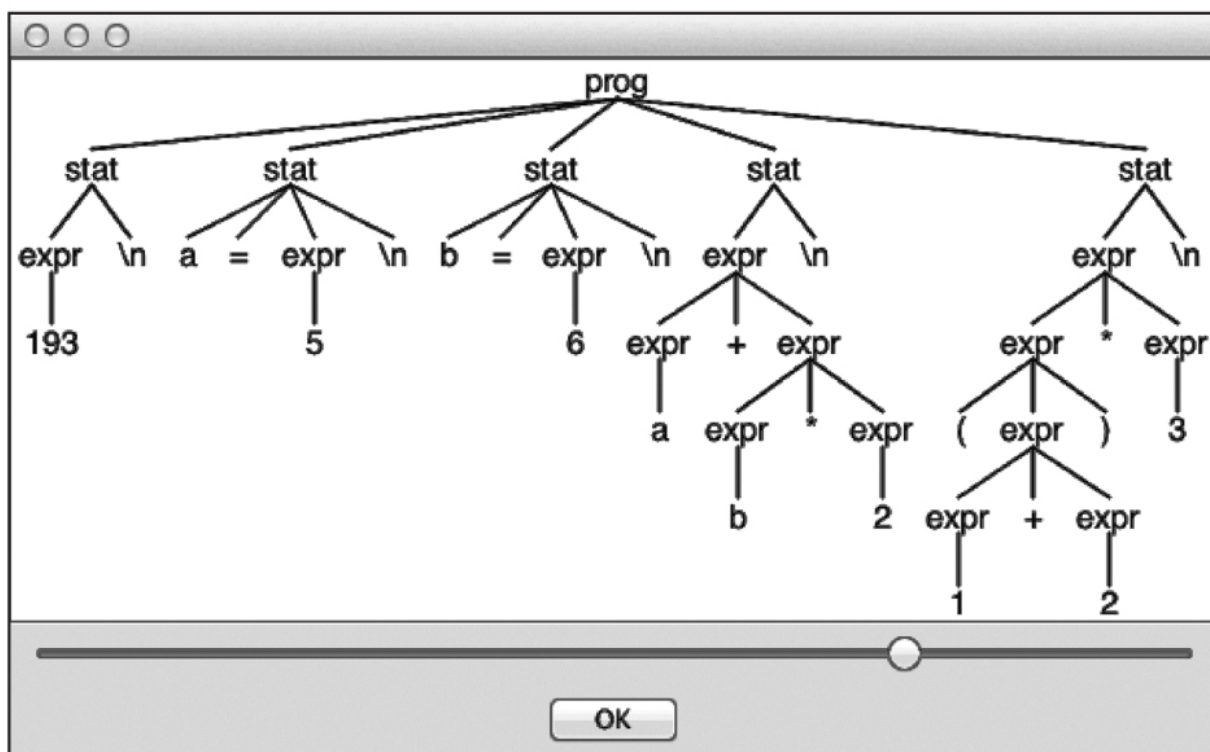


图4-1 显示语法分析树的窗口

语法分析树类似于我们的语法分析器在识别输入文本时的函数调用树（ANTLR为每条规则生成一个函数）。

使用测试组件开发和测试语法是一种不错的方法，不过最终，我们需要把ANTLR为我们自动生成的语法分析器集成到程序中。

下面的main程序展示了一些必要的代码，这些代码首先新建出所需的所有对象，然后针对prog规则启动我们的“表达式语言”语法分析器。

```
tour/ExprJoyRide.java
第1行 import org.antlr.v4.runtime.*;
- import org.antlr.v4.runtime.tree.*;
- import java.io.FileInputStream;
- import java.io.InputStream;
5 public class ExprJoyRide {
-     public static void main(String[] args) throws Exception {
-         String inputFile = null;
-         if ( args.length>0 ) inputFile = args[0];
-         InputStream is = System.in;
10        if ( inputFile!=null ) is = new FileInputStream(inputFile);
-         ANTLRInputStream input = new ANTLRInputStream(is);
-         ExprLexer lexer = new ExprLexer(input);
-         CommonTokenStream tokens = new CommonTokenStream(lexer);
-         ExprParser parser = new ExprParser(tokens);
15        ParseTree tree = parser.prog(); // 从 prog 规则开始进行语法分析
-         System.out.println(tree.toStringTree(parser)); // 以文本形式打印树
-     }
- }
```

第7行到第11行为词法分析器新建了一个处理字符的输入流。第12行到第14行新建了词法分析器和语法分析器对象，以及一个架设在二者之间的词法符号流“管道”。第15行真正启动了语法分析器，开始了解析过程（调用一条规则对应的方法就等于指定该规则开始语法分析，我们可以调用任何我们所希望的规则方法）。最后，第16行用文本形式将该规则方法prog（）返回的语法分析树打印出来。

下面是构建测试程序以及对输入的t.expr文件运行该程序的详细步骤：

```

⇒ $ javac ExprJoyRide.java Expr*.java
⇒ $ java ExprJoyRide t.expr
< (prog
  (stat (expr 193) \n)
  (stat a = (expr 5) \n)
  (stat b = (expr 6) \n)
  (stat (expr (expr a) + (expr (expr b) * (expr 2)))) \n)
  (stat (expr (expr ( (expr (expr 1) + (expr 2)) )) * (expr 3)) \n)
)

```

以这样的文本形式（稍加整理）展示的语法分析树显得不那么直观，不过对于功能测试来说，它还是非常有用的。我们这个“表达式语法”只有寥寥数行，但是实际中的语法可能多达成千上万行。在下一节中，我们将会学习如何使那样的大型语法维持在可控范围内。

1. 语法导入

一个好主意是，将非常大的语法拆分成逻辑单元，正如我们在软件开发中所做的那样。拆分的方法之一是将语法分为两部分：语法分析器的语法和词法分析器的语法。这是一个不错的方案，因为在不同语言的词法规则中，有相当大的一部分是重复的。例如，不同语言的标识符和数字定义通常是相同的。将词法规则重构并抽取出来成为一个“模块”意味着我们可以将它应用于不同的语法分析器。下面的这个词法语法包含了上面的“表达式语法”中所有的词法规则。

```
tour/CommonLexerRules.g4
```

```
lexer grammar CommonLexerRules; // 注意区别，是 "lexer grammar"

ID : [a-zA-Z]+ ; // 匹配标识符
INT : [0-9]+ ; // 匹配整数
NEWLINE: '\r'? '\n' ; // 告诉语法分析器一个新行的开始（即语句终止标志）
WS : [ \t]+ -> skip ; // 丢弃空白字符
```

现在我们可以将原先的语法中的那些词法规则替换为一个import语句了。

```
tour/LibExpr.g4
```

```
grammar LibExpr; // 为了和原先的语法区分开，进行了重命名
import CommonLexerRules; // 引入 CommonLexerRules.g4 中的全部规则
/** 起始规则，语法分析的起点 . */
prog: stat+ ;
stat: expr NEWLINE
    | ID '=' expr NEWLINE
    | NEWLINE
    ;

expr: expr ('*' | '/') expr
    | expr ('+' | '-') expr
    | INT
    | ID
    | '(' expr ')'
    ;
```

构建和测试过程与重构之前相同。我们不需要对被导入的语法运行ANTLR。

```

⇒ $ antlr4 LibExpr.g4 # 将会自动获取 CommonLexerRules.g4
⇒ $ ls Lib*.java
  < LibExprBaseListener.java      LibExprListener.java
    LibExprLexer.java            LibExprParser.java
⇒ $ javac LibExpr*.java
⇒ $ grun LibExpr prog -tree
⇒ 3+4
⇒ E0F
  < (prog (stat (expr (expr 3) + (expr 4)) \n))

```

到现在为止，我们假设输入都是合法的，但是错误处理是几乎所有语言类应用程序必不可少的部分。我们接下来将会看到，ANTLR如何处理有错误的输入。

2.处理有错误的输入

ANTLR语法分析器能够自动报告语法错误并从错误中恢复。例如，如果我们的表达式少了一个右括号，语法分析器将会自动输出一个错误信息。

```

⇒ $ java ExprJoyRide
⇒ (1+2
⇒ 3
⇒ E0F
  < line 1:4 mismatched input '\n' expecting {'}', '+', '*', '-', '/' }
    (prog
      (stat (expr ( (expr (expr 1) + (expr 2)) <missing '>'>) \n)
        (stat (expr 3) \n)
      )
    )

```

同样值得重视的是，语法分析器从错误中恢复，并且正确地完成了第二个表达式的匹配（表达式3）。

如果在grun命令中使用了“-gui”选项，语法分析树对话框会将错误节点自动标红，如图4-2所示。

```
⇒ $ grun LibExpr prog -gui  
⇒ (1+2  
⇒ 34*69  
⇒ EoF
```

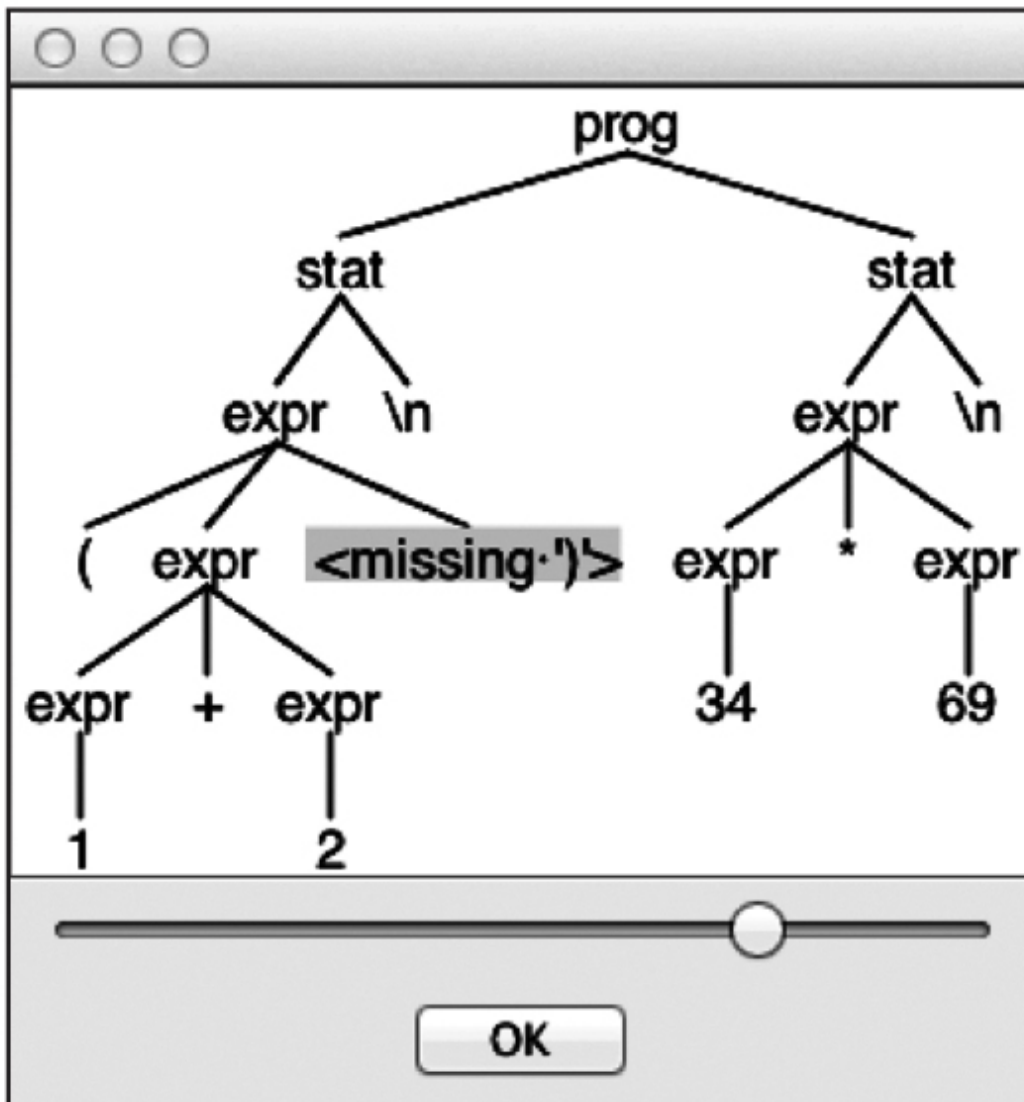


图4-2 被自动标红的语法分析树对话框

注意**ANTLR**成功地从第一个表达式的错误中恢复，正确地匹配了第二个表达式。

ANTLR的错误处理机制有很高的灵活性。我们可以修改输出的错误信息，捕获识别过程中的异常，甚至改变基本的异常处理策略。我们将在第9章中涉及这些内容。

以上就是有关语法和语法分析的快速而奇妙的旅程。我们看到了一个简单的“表达式语法”，以及使用内置的测试组件和**main**示例程序开始语法分析过程的方法。我们也看到了语法分析树的文本形式和可视化形式的展示，了解了我们的语法是如何匹配输入文本的。**import**语句赋予我们编写模块化语法的能力。接下来，让我们更进一步，不仅能够识别语言，还能完成解析表达式的工作，即计算表达式的值。

4.2 利用访问器构建一个计算器

为了能让上一节中的算术表达式语法分析器完成计算结果的工作，我们需要写一些**Java**代码。**ANTLR 4**推荐使用语法分析树访问器和其他的遍历器来实现语言类应用程序，从而保持语法本身的整洁。本节中，我们会使用广为人知的访问者模式来实现我们的小计算器。为了简化我们的工作，**ANTLR**自动生成了一个访问器接口和一个空的实现类。

在开始之前，我们需要先对语法做少量的修改。首先，我们需要给备选分支加上标签（这些标签可以是任意标识符，只要它们不与规则名

冲突)。如果备选分支上面没有标签，**ANTLR**就只为每条规则生成一个方法（第7章使用了一个相似的语法对访问器机制进行了详细讲解）。在本例中，我们希望每个备选分支都有不同的访问器方法，这样我们就可以对每种输入都获得一个不同的“事件”。在我们的新语法 **LabeledExpr** 中，标签以#开头，放置在一个备选分支的右侧。

tour/LabeledExpr.g4

```
stat:  expr NEWLINE          # printExpr
      | ID '=' expr NEWLINE  # assign
      | NEWLINE              # blank
      ;

expr:  expr op=('*' | '/') expr  # MulDiv
      | expr op=('+' | '-' ) expr # AddSub
      | INT                     # int
      | ID                      # id
      | '(' expr ')'            # parens
      ;
```

接下来，我们为运算符这样词法符号定义一些名字，这样，在随后的访问器中，我们就可以将这些词法符号的名字当作**Java**常量来引用。

tour/LabeledExpr.g4

```
MUL :  '*' ; // 为上述语法中使用的 '*' 命名
DIV :  '/' ;
ADD :  '+' ;
SUB :  '-' ;
```

在完成这份增强版的语法之后，我们就可以开始编码实现我们的计算器了。可以看到，**Calc.java**的**main ()**方法几乎和之前的**ExprJoyRide.java**完全一样。差别之一是，在新程序中，我们创建的词

法分析器对象和语法分析器对象是基于语法LabeledExpr的，而非Expr。

tour/Calc.java

```
LabeledExprLexer lexer = new LabeledExprLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
LabeledExprParser parser = new LabeledExprParser(tokens);
ParseTree tree = parser.prog(); // 开始语法分析
```

此外，我们还去掉了以文本方式打印语法分析树的语句。另外一个差别是我们实例化了一个自定义的访问器——EvalVistor，稍后我们将详细分析它。我们调用visit () 方法，开始遍历prog () 方法返回的语法分析树。

tour/Calc.java

```
EvalVisitor eval = new EvalVisitor();
eval.visit(tree);
```

现在已经万事俱备了，我们只需要实现一个能够遍历语法分析树、计算并返回结果的访问器即可。在动手之前，我们先来看看ANTLR能为我们自动生成哪些东西。

⇒ **\$ antlr4 -no-listener -visitor LabeledExpr.g4**

首先，ANTLR自动生成了一个访问器接口，并为其中每个带标签的备选分支生成了一个方法。


```

public interface LabeledExprVisitor<T> {
    T visitId(LabeledExprParser.IdContext ctx);          # 来自标签 id
    T visitAssign(LabeledExprParser.AssignContext ctx); # 来自标签 assign
    T visitMulDiv(LabeledExprParser.MulDivContext ctx); # 来自标签 MulDiv
    ...
}

```

该接口使用了Java的泛型定义，参数化的类型是visit方法的返回值的类型。这允许我们的实现类使用自定义的返回值类型，以适应不同的场合。

其次，ANTLR生成了该访问器的一个默认实现类

LabeledExprBaseVisitor供我们使用。考虑我们的实际情况，表达式的计算结果都是整数，因此我们的EvalVistor类应该继承

LabeledExprBaseVisitor<Integer>类。为最终完成计算器的实现，我们覆盖了访问器中表达式和赋值语句规则对应的方法。下面是该类的完整代码。

```

tour/EvalVisitor.java
import java.util.HashMap;
import java.util.Map;

```

```

public class EvalVisitor extends LabeledExprBaseVisitor<Integer> {
    /** 计算器的“内存”，存放变量名和变量值的对应关系 */
    Map<String, Integer> memory = new HashMap<String, Integer>();

    /** ID '=' expr NEWLINE */
    @Override
    public Integer visitAssign(LabeledExprParser.AssignContext ctx) {
        String id = ctx.ID().getText(); // id在'='的左侧
        int value = visit(ctx.expr()); // 计算右侧表达式的值
        memory.put(id, value); // 将这个映射关系存储在计算器的“内存”中
        return value;
    }

    /** expr NEWLINE */
    @Override
    public Integer visitPrintExpr(LabeledExprParser.PrintExprContext ctx) {
        Integer value = visit(ctx.expr()); // 计算expr子节点的值
        System.out.println(value); // 打印结果
        return 0; // 上面已经直接打印出了结果，因此这里返回一个假值即可
    }

    /** INT */
    @Override
    public Integer visitInt(LabeledExprParser.IntContext ctx) {
        return Integer.valueOf(ctx.INT().getText());
    }

    /** ID */
    @Override
    public Integer visitId(LabeledExprParser.IdContext ctx) {
        String id = ctx.ID().getText();
        if ( memory.containsKey(id) ) return memory.get(id);
        return 0;
    }

    /** expr op=('*' | '/') expr */
    @Override
    public Integer visitMulDiv(LabeledExprParser.MulDivContext ctx) {
        int left = visit(ctx.expr(0)); // 计算左侧子表达式的值
        int right = visit(ctx.expr(1)); // 计算右侧子表达式的值
        if ( ctx.op.getType() == LabeledExprParser.MUL ) return left * right;
        return left / right; // 如果不是乘法，就一定是除法
    }

    /** expr op=('+' | '-') expr */
    @Override
    public Integer visitAddSub(LabeledExprParser.AddSubContext ctx) {
        int left = visit(ctx.expr(0)); // 计算左侧子表达式的值
        int right = visit(ctx.expr(1)); // 计算右侧子表达式的值
        if ( ctx.op.getType() == LabeledExprParser.ADD ) return left + right;
    }
}

```

```

        return left - right; // 如果不是加法，就一定是减法
    }

    /** '(' expr ')' */
    @Override
    public Integer visitParens(LabeledExprParser.ParensContext ctx) {
        return visit(ctx.expr()); // 返回子表达式的值
    }
}

```

下面是以上程序的构建和测试过程，使用t.expr作为输入：

```

⇒ $ antlr4 -no-listener -visitor LabeledExpr.g4 # 必须加 -visitor 参数 !!!
⇒ $ ls LabeledExpr*.java
  LabeledExprBaseVisitor.java      LabeledExprParser.java
  LabeledExprLexer.java           LabeledExprVisitor.java
⇒ $ javac Calc.java LabeledExpr*.java
⇒ $ cat t.expr
  193
  a = 5
  b = 6
  a+b*2
  (1+2)*3
⇒ $ java Calc t.expr
  193
  17
  9

```

上述计算器的构建过程透露出一个信息：我们不需要像ANTLR 3那样，在语法文件中插入Java代码编写的动作（action）。语法文件独立于程序，具有编程语言中立性。访问器机制也使得一切语言识别之外的工作在我们所熟悉的Java领域进行。在生成的所需的语法分析器之后，就不再需要同ANTLR语法标记打交道了。

在继续学习之前，你可能需要花费一点工夫，给我们的“表达式语言”增加一个`clear`语句。这是一个锻炼你的好机会，让你亲自动手进行实际操作，而又无须深入了解全部细节。`clear`命令会将计算器的“内存”清空（即`EvalVisitor`的`memory`成员），你需要在`stat`规则中增加一个新备选分支来识别它。使用`#clear`来给这个新的备选分支加上标签，然后对修改后的语法运行`ANTLR`命令，获得生成的访问器接口。然后，为了能在接收`clear`命令的时候作出响应，你需要实现`visitClear()`方法。最后，按照之前的步骤编译并运行`Calc`。

接下来，让我们换个主题，考虑一下进行语言的翻译，而不仅仅是对输入进行求值和语义解释。下一节中，我们将会使用另外一种机制（即监听器）来构建一个针对`Java`源代码的翻译程序。

4.3 利用监听器构建一个翻译程序

想象一下，你的老板让你编写一个工具，用来将一个`Java`类中的全部方法抽取出来，生成一个接口文件。如果你是个新手，这自然会引起你的恐慌。如果你是个经验丰富的`Java`开发者，你可能会使用`Java`反射API或者`javap`工具从`Java`类中提取方法的签名。如果你的`Java`功力深厚，你甚至可以使用字节码库`ASM`来完成工作。紧接着，你的老板说：“对了，记得保留方法签名中的空白字符和注释。”现在已经别无选择，我们必须解析`Java`源代码了。例如，我们想要读取这样的`Java`源代码：

```
tour/Demo.java
import java.util.List;
import java.util.Map;
public class Demo {
    void f(int x, String y) { }
    int[ ] g(/*no args*/) { return null; }
    List<Map<String, Integer>>[] h() { return null; }
}
```

然后使用其中的方法签名生成一个接口，保留其中的全部空白字符和注释。

```
tour/IDemo.java
interface IDemo {
    void f(int x, String y);
    int[ ] g(/*no args*/);
    List<Map<String, Integer>>[] h();
}
```

信不信由你，我们能够用大约15行代码解决这个问题。这些代码是通过监听Java语法分析树遍历器触发的“事件”来完成这项工作的。Java语法分析树是由解析Java语言的语法分析器生成的，本书的源代码中提供了Java语言的ANTLR语法。我们将会从类定义中提取类名，用它来命名生成的接口，然后从类的方法定义中获取方法签名（返回值、方法名，以及参数列表）。在8.3节中有一个相似的、解释更加详细的例子。

在Java语言语法和我们的监听器对象间的关键“接口”是JavaListener，是ANTLR为我们自动生成的一个类。它定义了ANTLR的运行库中的ParseTreeWalker类在遍历语法分析树时能够触发的全部方法。在本例中，我们需要通过覆盖对应的方法，对三个事件作出响应：遍历器进

入和离开类定义时，以及遍历器遇到方法定义时。下面是生成的监听器接口中的相关方法：

```
public interface JavaListener extends ParseTreeListener {
    void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx);
    void exitClassDeclaration(JavaParser.ClassDeclarationContext ctx);
    void enterMethodDeclaration(JavaParser.MethodDeclarationContext ctx);
    ...
}
```

访问器机制和监听器机制的最大的区别在于，监听器的方法会被ANTLR提供的遍历器对象自动调用，而在访问器的方法中，必须显式调用visit方法来访问子节点。忘记调用visit（）的后果就是对应的子树将不会被访问。为实现我们自己的监听器，我们需要知道classDeclaration规则和methodDeclaration规则长什么样子，因为监听器方法需要获取这些规则匹配的词组元素。Java.g4是一个完整的Java语法，这里只截取我们解决问题所需的两个片段。

tour/Java.g4

```
classDeclaration
    : 'class' Identifier typeParameters? ('extends' type)?
      ('implements' typeList)?
      classBody
    ;
```

tour/Java.g4

```
methodDeclaration
    : type Identifier formalParameters ('[' ']* methodDeclarationRest
    | 'void' Identifier formalParameters methodDeclarationRest
    ;
```

我们无需实现全部实现接口中200个左右的方法，因为ANTLR为我们生成了一个默认的名为JavaBaseListener的实现。我们的“接口提取器”可以

继承JavaBaseListener，覆盖我们所感兴趣的那些方法。

我们的基本思想是，在类定义的起始位置打印出接口定义，然后在类定义的结束位置打印出}。在遇到每个方法定义时，我们将会抽取出它的签名。下面是完整的实现代码：

```
tour/ExtractInterfaceListener.java
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.misc.Interval;

public class ExtractInterfaceListener extends JavaBaseListener {
    JavaParser parser;
    public ExtractInterfaceListener(JavaParser parser) {this.parser = parser;}
    /** 监听对类定义的匹配 */
    @Override
    public void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx){
        System.out.println("interface I"+ctx.Identifier()+" {");
    }
    @Override
    public void exitClassDeclaration(JavaParser.ClassDeclarationContext ctx) {
        System.out.println("}");
    }
    /** 监听对方法定义的匹配 */
    @Override
    public void enterMethodDeclaration(
        JavaParser.MethodDeclarationContext ctx
    )
    {
        // 需要从语法分析器中获取词法符号
        TokenStream tokens = parser.getTokenStream();
        String type = "void";

        if ( ctx.type()!=null ) {
            type = tokens.getText(ctx.type());
        }
        String args = tokens.getText(ctx.formalParameters());
        System.out.println("\t"+type+" "+ctx.Identifier()+args+");");
    }
}
```


我们需要一个main程序来执行上面的代码，其内容和本章之前的那些main程序几乎相同。在我们启动语法分析器后，上述代码就会被执行。

```
tour/ExtractInterfaceTool.java
```

```
JavaLexer lexer = new JavaLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
JavaParser parser = new JavaParser(tokens);
ParseTree tree = parser.compilationUnit(); // 开始语法分析的过程

ParseTreeWalker walker = new ParseTreeWalker(); // 新建一个标准的遍历器
ExtractInterfaceListener extractor = new ExtractInterfaceListener(parser);
walker.walk(extractor, tree); // 使用监听器初始化对语法分析树的遍历
```

记得在这个文件的开头添加“import org.antlr.v4.runtime.tree.*; ”。当准备好Java.g4和我们的ExtractInterfaceTool中的main () 方法之后，下面是完整的构建和测试步骤：

```
⇒ $ antlr4 Java.g4
⇒ $ ls Java*.java ExtractInterface*.java
  ExtractInterfaceListener.java  JavaBaseListener.java  JavaListener.java
    ExtractInterfaceTool.java    JavaLexer.java         JavaParser.java
⇒ $ javac Java*.java Extract*.java
⇒ $ java ExtractInterfaceTool Demo.java
  < interface IDemo {
        void f(int x, String y);
        int[ ] g(/*no args*/);
        List<Map<String, Integer>>[] h();
    }
```

我们实现的这个“接口提取器”功能并不完整，因为它没有为接口定义添加原有类中的import语句，生成的接口可能引用了这些import语句所对应的类型，例如List。作为练习，请你试着处理一下import语句。这会使你确信，使用监听器机制来构建这种提取器或者翻译器是如此容

易。我们甚至不需要知道`importDeclaration`规则长什么样子，因为在`enterImportDeclaration ()`方法中，只需要简单地打印出整条规则匹配的文本即可：`parser.getTokenStream ().getText (ctx)`。

访问器和监听器机制表现出色，它们使语法分析过程和程序本身高度分离。尽管如此，有些时候，我们还是需要额外的灵活性和可操控性。

4.4 定制语法分析过程

监听器和访问器机制是一个创举，这使得自定义的程序代码和语法本身分离开来，让语法更具可读性，避免了将语法和特定的程序混杂在一起。不过，为了极佳的灵活性和可操控性，我们可以直接将代码片段（动作）嵌入语法中。这些动作将被拷贝到ANTLR自动生成的递归下降语法分析器的代码中。本节中，我们将实现一个简单的程序，读取若干行数据，然后将指定列的值打印出来。之后，我们将会看到如何实现特殊的动作，叫作语义判定（`semantic predicate`），它能够动态地开启或者关闭部分语法。

1.在语法中嵌入任意动作

如果不想承担建立语法分析树的开销，我们可以在语法分析的过程中计算并打印结果。另一个方案是，在“表达式语法”中嵌入一些代码。

这种方案难度更高，因为我们必须知道嵌入的动作对语法分析器的影响，以及在哪里放置这些动作。

为展示如何在语法中嵌入动作，我们来构建一个能够打印若干行数据中的指定列的程序。我一直想完成一个这样的程序，因为大家总是给我发一些文本文件，我想要从中提取特定的列，如名字或者电子邮箱地址。例如，我们拥有以下数据：

tour/t.rows		
parrt	Terence Parr	101
tombu	Tom Burns	020
bke	Kevin Edgar	008

列之间是用`tab`符分隔的，每行以一个换行符结尾。匹配这样的输入文件的语法非常简单：

```
file : (row NL)+ ; // NL 是换行符 : '\r'? '\n'
row  : STUFF+ ;
```

当我们加入动作时，上述语法就会变得混乱。我们需要在其中创建一个构造器，这样我们就能传入希望提取的列号（从1开始计数）；另外，我们需要在`row`规则的“`(...)+`”循环中放置一些动作。

```
tour/Rows.g4
```

```
grammar Rows;
```

```
@parser::members { // 在生成的 RowsParser 中添加一些成员
    int col;
    public RowsParser(TokenStream input, int col) { // 自定义的构造器
        this(input);
        this.col = col;
    }
}

file: (row NL)+ ;

row
locals [int i=0]
    : (    STUFF
        {

            $i++;
            if ( $i == col ) System.out.println($STUFF.text);
        }

    )+
    ;

TAB  : '\t' -> skip ;    // 匹配但是不将其传递给语法分析器
NL   : '\r'? '\n' ;     // 匹配并将其传递给语法分析器
STUFF: ~[\t\r\n]+ ;     // 匹配除 tab 符和换行符之外的任何字符
```

STUFF词法规则匹配除tab符和换行符之外的任何字符，这意味着数据中可以包含空格。

到现在为止，你应该已经很熟悉一个适当的main程序了。下面程序的唯一不同之处在于，我们给语法分析器的自定义构造器传递了一个列号，并且告诉语法分析树不必建立语法分析树。

tour/Col.java

```
RowsLexer lexer = new RowsLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
int col = Integer.valueOf(args[0]);
RowsParser parser = new RowsParser(tokens, col); // 传递列号作为参数
parser.setBuildParseTree(false); // 不需要浪费时间建立语法分析树
parser.file(); // 开始语法分析
```

其中的很多细节我们将在第10章中深入探究。现在看来，动作就是花括号包围的一些代码片段。**members**动作可以将代码注入到生成的语法分析器类中，使之成为该类的成员。在**row**规则中的动作访问了**\$i**，它是一个使用**locals**子句定义的局部变量。**row**规则也使用了**\$STUFF.text**来获得刚刚匹配的**STUFF**词法符号中包含的文本。

下面是构建和测试的步骤，每列进行一次测试。

```
⇒ $ antlr4 -no-listener Rows.g4 # 不需要生成监听器
⇒ $ javac Rows*.java Col.java
⇒ $ java Col 1 < t.rows          # 从文件 t.rows 中读取并打印第一列
  < parrt
    tombu
    bke
⇒ $ java Col 2 < t.rows
  < Terence Parr
    Tom Burns
    Kevin Edgar
⇒ $ java Col 3 < t.rows
  < 101
    020
    008
```

这些动作在语法分析器的匹配过程中提取并打印相关的值，但是并不改变语法分析过程本身。除此之外，动作还能改变语法分析器识别输入文本的过程。在下一节中，我们将进一步理解内嵌动作的概念。

2.使用语义判定改变语法分析过程

我们会在第11章中通过一个简单的例子来展示语义判定的威力。在此之前，让我们先来看一个读取一系列整数的语法。它耍了一个小把戏：其中的一部分整数指定了接下来的多少个整数分为一组。下面是样例输入：

```
tour/t.data
2 9 10 3 1 2 3
```

第一个数字2告诉我们，匹配接下来的两个数字9和10。紧接着的数字3告诉我们匹配接下来的三个数字。我们的目标是创建一份名为Data的语法，将9和10分为一组，然后1，2，3分为一组，像是下面这样：

```
⇒ $ antlr4 -no-listener Data.g4
⇒ $ javac Data*.java
⇒ $ grun Data file -tree t.data
  ( (file (group 2 (sequence 9 10)) (group 3 (sequence 1 2 3))) )
```

如图4-3所示的语法分析树清楚地显示了匹配到的分组。

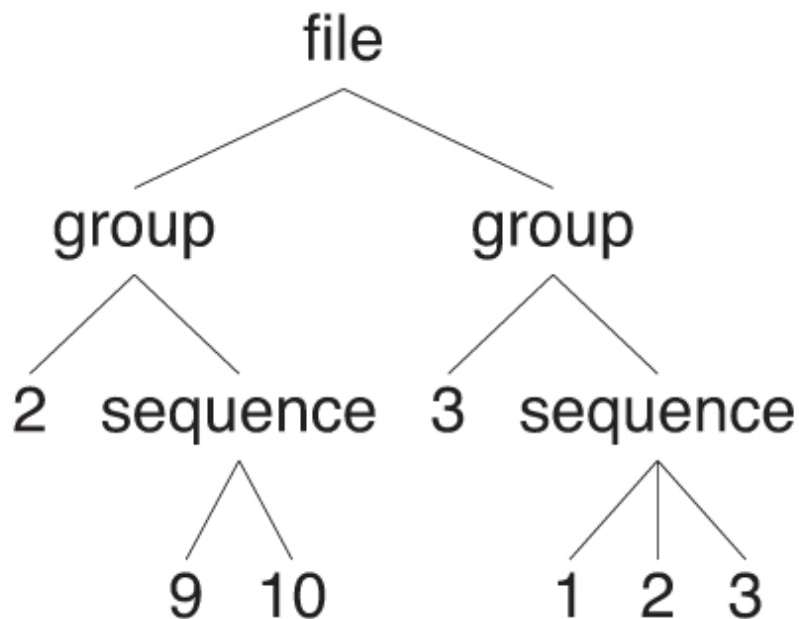


图4-3 显示了匹配到的分组的语法分析树

下面将看到的Data语法的关键在于一段动作，它的值是布尔类型的，称为一个语义判定： $\{ \$i \leq \$n \} ?$ 。它的值在匹配到n个输入整数之前保持为true，其中n是sequence语法中的参数。当语义判定的值为false时，对应的备选分支就从语法中“消失”了，因此，它也就从生成的语法分析器中“消失”了。在本例中，语义判定的值为false使得“(...) *”循环终止，从sequence规则返回。

```

tour/Data.g4
grammar Data;

file : group+ ;

group: INT sequence[$INT.int] ;

sequence[int n]
locals [int i = 1;]
    : ( {$i<=$n}? INT {$i++;} ) * // 匹配 n 个整数
    ;

INT : [0-9]+ ; // 匹配整数
WS : [ \t\n\r]+ -> skip ; // 丢弃所有的空白字符

```

语法分析器内部使用的sequence规则的可视化展示大致如图4-4所示。

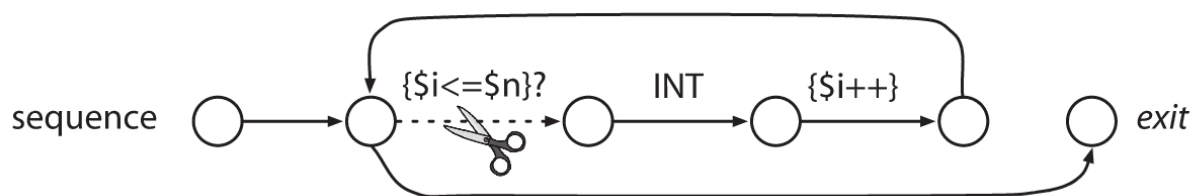


图4-4 sequence规则的可视化展示

剪刀和虚线显示语义判定会剪掉该路径，让语法分析器只剩一个可选路径：退出。

大多数情况下我们不需要如此精细的操作，不过知道有这样一件处理异常问题的利器总是好的。

迄今为止，我们的侧重点都是语法分析的功能，实际上，在词法分析的层面上还有很多有用的功能等着我们去发现，下面就让我们一起来看一看。

4.5 神奇的词法分析特性

ANTLR有三个与词法符号有关非常棒的特性，值得付诸笔墨。首先，我们将会尝试处理XML这样的具有不同词法结构的输入格式（标签内外不同）。其次，我们将会学习通过修改输入的词法符号流，在Java类中插入一个字段的方法。它将会展示，如何以最低的代价来生成和输入内容相似的输出。最后，我们将会看到ANTLR语法分析器如何忽略空白字符和注释，同时不丢弃它们。

1. 孤岛语法：处理相同文件中的不同格式

迄今为止，我们看到的样例输入文件都只包含一种语言，但是事实上，有很多常见的文件格式包含多重语言。例如，Java文档注释中的@author标签等内容使用的是一种特殊的微型语言；在注释之外的一切内容都是Java代码。类似StringTemplet和Django的模板引擎也存在相似的问题。它们必须将模板语言表达式之外的文本按照不同的方式进行处理。这种情况通常称为孤岛语法。

ANTLR提供了一个众所周知的词法分析器特性，称为词法分析模式（lexical mode），使我们能够方便地处理混杂着不同格式数据的文件。它的基本思想是，当词法分析器看到一些特殊的“哨兵”字符序列时，执行不同模式的切换。

XML是个很好的例子。一个XML解析器将除了标签和实体转义（例如£）之外的东西全部当作普通文本。当看到<时，词法分析器会切换到“标签内部”模式；当看到>或者/>时，它就切换回默认模式。下面的语法展示了XML解析器的工作方式。我们将在第12章中对它进行详细讨论。

```
tour/XMLLexer.g4
lexer grammar XMLLexer;

// 默认的“模式”：所有在标签之外的东西
OPEN      :   '<'                -> pushMode(INSIDE) ;
COMMENT    :   '<!--' .*? '-->'  -> skip ;
EntityRef  :   '&' [a-z]+ ';' ;
TEXT       :   ~( '<' | '&' )+ ;           // 匹配任意除 < 和 & 之外的 16 位字符

// ----- 所有在标签之内的东西 -----
mode INSIDE;
CLOSE      :   '>'                -> popMode ; // 回到默认模式
SLASH_CLOSE :   '/'>'            -> popMode ;
EQUALS     :   '=' ;
STRING     :   '"' .*? '"' ;
SlashName  :   '/' Name ;
Name       :   ALPHA (ALPHA|DIGIT)* ;
S          :   [ \t\r\n]         -> skip ;

fragment
ALPHA      :   [a-zA-Z] ;

fragment
DIGIT      :   [0-9] ;
```

下面是该语法的样例输入：

```
tour/t.xml
<tools>
    <tool name="ANTLR">A parser generator</tool>
</tools>
```

下面是构建和测试的步骤:

```
⇒ $ antlr4 XMLLexer.g4
⇒ $ javac XML*.java
⇒ $ grun XML tokens -tokens t.xml
< [@0,0:0='<,<1>,1:0]
  [@1,1:5='tools',<10>,1:1]
  [@2,6:6='>',<5>,1:6]
  [@3,7:8='\n\t',<4>,1:7]
  [@4,9:9='<,<1>,2:1]
  [@5,10:13='tool',<10>,2:2]
  [@6,15:18='name',<10>,2:7]
  [@7,19:19='',<7>,2:11]
  [@8,20:26='"ANTLR"',<8>,2:12]
  [@9,27:27='>',<5>,2:19]
  [@10,28:45='A parser generator',<4>,2:20]
  [@11,46:46='<,<1>,2:38]
  [@12,47:51='/tool',<9>,2:39]
  [@13,52:52='>',<5>,2:44]
  [@14,53:53='\n',<4>,2:45]

  [@15,54:54='<,<1>,3:0]
  [@16,55:60='/tools',<9>,3:1]
  [@17,61:61='>',<5>,3:7]
  [@18,62:62='\n',<4>,3:8]
  [@19,63:62='<EOF>',<-1>,4:9]
```

输出的每一行代表一个词法符号，它包括词法符号的序号、起始和终止的字符位置、内容文本，以及行号和行内位置。这些内容让我们了解了词法分析器将输入文本转换为词法符号的过程。

在上述启动测试组件的命令行中，使用的参数是**XML tokens**，在正常情况下，这里应该是一个语法名加一个起始规则名。如果需要令测试组件只运行词法分析器而不运行语法分析器，我们可以指定参数为语

法名加上一个特殊的规则名`tokens`。最后，我们使用了一个参数“`-tokens`”命令测试组件打印出匹配到的词法符号。

知道词法符号是如何从词法分析器流向语法分析器的是非常有用的。例如，一些与翻译相关的问题实际上就是对输入的修改。有时候，我们可以通过修改原先的词法符号流来达到目的，而不需要产生新的输出。

2. 重写输入流

接下来，让我们构建一个小工具，它能够修改Java源代码并插入`java.io.Serializable`使用的序列化版本标识符（`serialVersionUID`，类似Eclipse的自动生成功能）。我们不希望小题大做，仅仅为了这样一件小事（读取输入，稍事修改后输出）就把ANTLR根据Java语法生成的`JavaListener`接口中的方法全部实现。更简单的做法是，在原先的词法符号流中插入一个适当的代表常量字段的词法符号，然后打印出修改后的输入流。对症下药，才能事半功倍。

我们的`main`程序和4.3节中的`ExtractInterfaceTool.java`非常相似，除了其中的一部分：当遍历结束后，我们将词法符号流打印出来（箭头处）。

```
tour/InsertSerialID.java
```

```
ParseTreeWalker walker = new ParseTreeWalker(); // 新建一个标准的遍历器  
InsertSerialIDListener extractor = new InsertSerialIDListener(tokens);  
walker.walk(extractor, tree); // 使用监听器初始化对语法分析树的遍历
```

```
// 打印出修改后的词法符号流
```

```
➤ System.out.println(extractor.rewriter.getText());
```

在监听器的实现中，我们需要在类定义的起始位置触发一个插入操作：

```
tour/InsertSerialIDListener.java
```

```
import org.antlr.v4.runtime.TokenStream;  
import org.antlr.v4.runtime.TokenStreamRewriter;  
  
public class InsertSerialIDListener extends JavaBaseListener {  
    TokenStreamRewriter rewriter;  
    public InsertSerialIDListener(TokenStream tokens) {  
        rewriter = new TokenStreamRewriter(tokens);  
    }  
    @Override  
    public void enterClassBody(JavaParser.ClassBodyContext ctx) {  
        String field = "\n\tpublic static final long serialVersionUID = 1L;";  
        rewriter.insertAfter(ctx.start, field);  
    }  
}
```

其中的关键之处在于，`TokenStreamRewriter`对象实际上修改的是词法符号流的“视图”而非词法符号流本身。它认为所有对修改方法的调用都只是一个“指令”，然后将这些修改放入一个队列；在未来词法符号流被重新渲染为文本时，这些修改才会被执行。在每次我们调用`getText()`的时候，`rewriter`对象都会执行上述队列中的指令。

让我们用之前用过的`Demo.java`文件来测试上面的代码。

```

⇒ $ antlr4 Java.g4
⇒ $ javac InsertSerialID*.java Java*.java
⇒ $ java InsertSerialID Demo.java
< import java.util.List;
   import java.util.Map;
   public class Demo {
       public static final long serialVersionUID = 1L;
       void f(int x, String y) { }
       int[ ] g(/*no args*/) { return null; }
       List<Map<String, Integer>>[] h() { return null; }
   }

```

通过寥寥数行代码，我们就完成了对Java类定义的修改，同时又不影响我们插入位置之外的任何代码。这样的策略在源代码插桩或者重构等场合下是非常有效的。**TokenStreamRewriter**能够修改词法符号流，是一个非常高效和强大的工具。

在结束本章之前，还有一些非常赞的内容要介绍。这部分内容包含一个很常见的问题，如果没有**ANTLR**的词法符号通道机制，它将会变得非常棘手。

3.将词法符号送入不同通道

之前我们看到，**Java**接口抽取器魔术般地保留了方法签名中的空白字符和注释，如下所示：

```
int[ ] g(/*no args*/) { return null; }
```

使用传统方法很难达到这个目的。对于大多数语法，语法分析器是可以忽略空白字符和注释的。如果我们不想让空白字符和注释在语法中

到处都是，我们就必须让词法分析器丢弃它们。不幸的是，这意味着程序中将完全无法访问空白字符和注释，也无法对它们进行进一步处理。忽略却保留注释和空白字符的秘诀是将这些词法符号送入一个“隐藏通道”。语法分析器只处理一个通道，因此我们可以将希望保留的词法符号送入其他通道内。下面是完成上述工作的Java语法：

```
tour/Java.g4
COMMENT
    :   '/*' .*? '*/'      -> channel(HIDDEN) // 匹配 /* 和 */ 之间的任何东西
    ;
WS    :   [ \r\t\u000C\u00A0]+ -> channel(HIDDEN)
    ;
```

同我们之前讨论过的“->skip”一样，“->channel (HIDDEN)”也是一个词法分析器指令。此处，它设置了这些词法符号的通道号，这样，这些词法符号就会被语法分析器忽略。词法符号流中仍然保存着这些原始的词法符号序列，只不过在向语法分析器提供数据时忽略了那些处于已关闭通道的词法符号。

介绍完上面这些词法分析器的特性，我们就圆满地完成了今天的ANTLR之旅。本章介绍了使ANTLR灵活易用的基本要素。我们并没有涉及任何细节，而是看到了，在实践中，ANTLR是如何解决一些不大但却现实的问题的。我们感受了ANTLR的语法符号。我们实现的访问器和监听器让我们不需要在语法中嵌入动作就能完成计算和翻译工作。我们也看到了，有些时候，内嵌动作是进行特殊的内部控制必不可少

可少的手段。最后，我们看到了一些词法分析器和词法符号流的神奇特性。

接下来，我们需要做的是，放慢脚步，带着一颗求知若渴的心重新审视本章中的概念。下一部分中的每一章都让我们离成为编程语言的实现者更近一步。我们将从学习ANTLR语法标记开始，逐步了解如何参照示例和编程语言的参考手册编写ANTLR语法。在拥有这些基础知识后，我们将会为真实世界中的编程语言编写ANTLR语法，然后深入学习之前简单提及的语法分析树监听器和访问器机制。

之后，在第三部分中，我们将会研究一些大师级的主题。

本书由“ePUBw.COM”整理，ePUBw.COM 提供
最新最全的优质电子书下载！！！！

第二部分 ANTLR开发语言类应用程序

在第二部分中，我们将会学习如何参照编程语言的标准和样例输入来编写语法。我们将会为CSV格式（comma-separated value）、JSON、

DOT图形格式（一种简单的编程语言）以及R语言构造语法。之后，我们将通过遍历语法分析树来深入研究构建语言类应用程序的细节。

第5章 设计语法

在第一部分中，我们认识了**ANTLR**，也了解了语法和语言类应用程序。现在，我们要放慢脚步，学习一些实用的细节，例如建立内部数据结构，提取信息，以及翻译输入内容等。我们的第一步是学习如何编写语法。本章中，我们将分析编程语言中最常见的语法结构和词法结构，学会如何用**ANTLR**标记来表达它们。在此基础上，我们将在下一章中构造一些真实的语法。

我们不能仅仅通过一些晦涩难懂的**ANTLR**概念来学习构造语法。首先，我们需要研究编程语言的通用模式，学习如何在语句中将它们辨识出来。通过这种研究，我们就能大体上得知这种语言的结构（一种语言模式就是一种递归的语法结构，例如，英语的一个句子包含“主语-谓语动词-宾语”，而日语的一个句子包含“主语-宾语-谓语动词”）。最终，我们需要从一系列有代表性的输入文件中归纳出一门语言的结构。在完成这样的归纳工作后，我们就可以正式使用**ANTLR**语法来表达这门语言了。

好消息是，虽然人们在过去的五十年间发明了很多种编程语言，但是，相对而言，基本的语言模式种类并不多。这种情况的出现，是因

为人们在设计编程语言时，倾向于将它们设计的和脑海中的自然语言相似。我们期望看到有序的词法符号，也期望看到词法符号间的依赖关系。例如， $\{ \}$ 是不符合语法的，因为其中的词法符号顺序不对； $(1+2$ 因为少了一个配对的) 而令人难以接受。除此之外，编程语言通常也因设计者使用了通用的数学符号而显得十分相似。在词法层面上，不同的编程语言也倾向于使用相同的结构，例如标识符、整数、字符串，等等。

对单词顺序和单词间依赖关系的限制来源于自然语言，逐渐发展为以下四种抽象的计算机语言模式：

- 序列：即一系列元素，例如一个数组初始化语句中的值。
- 选择：在多种可选方案中做出选择，例如编程语言中的不同种类的语句。
- 词法符号依赖：一个词法符号需要和某处的另外一个词法符号配对，例如左右括号匹配。
- 嵌套结构：一种自相似的语言结构，例如编程语言中的嵌套算术表达式或者嵌套语句块。

为实现以上模式，我们的语法规则只需要可选方案、词法符号引用和规则引用即可（巴克斯-诺尔范式，**Backus-Naur-Format**，**BNF**）。尽管

如此，为方便起见，我们还是将这些元素划分为子规则。子规则是用括号包围的内联规则。我们可以用以下符号标记子规则，用于指明其中的语法片段出现的次数：可选（?）、出现0次或多次（*）、至少一次（+）（扩展巴克斯-诺尔范式，**Extended Backus-Naur-Format**, **EBNF**）。

大多数读者应该曾经见过这种语法的形式，或者至少曾经接触过正则表达式，不过，为确保所有人在同一起点上，我们还是从最基础的部分讲起。

5.1 从编程语言的范例代码中提取语法

编写语法和编写软件很相似，差异在于我们处理的是语言规则，而非函数或者过程（**procedure**）。（记住，**ANTLR**将会为你的语法中的每条规则生成一个函数。）不过，在深入研究语法的细节之前，一件大有裨益的事情是：讨论语法的整体结构以及如何建立初始的语法框架。这就是我们本节要完成的事情，因为它是任何编程语言项目的基础步骤。如果你迫不及待地要构建和执行你的第一个语法分析器，不妨回顾第4章，或者跳到下一章去看第一个例子。在我们学习基本知识的过程中，可能会在下一章的例子里来回跳跃，所以请做好准备。

语法由一个为该语法命名的头部定义和一系列可以相互引用的语言规则组成。

```
grammar MyG;  
rule1 : <<stuff>> ;  
rule2 : <<more stuff>> ;  
...
```

和编写软件一样，我们必须指明我们需要的语言规则，即其中<<stuff>>的具体内容，以及哪条规则是起始规则（好比是main（）方法）。

为了给某种编程语言编写语法，我们必须要么精通它，要么拥有很多有代表性的、由该语言所编写的样例程序。显然，如果该语言的参考手册中给出了语法，或者存在另外一种自动生成器对其语法进行的表述，那就再好不过了。不过，就现在而言，让我们假设我们没有现成的语法作为参考。

设计良好的语法反映了编程世界中的功能分解或者自顶向下的设计。这意味着我们对语言结构的辨识是从最粗的粒度开始，一直进行到最详细的层次，并把它们编写成为语法规则。所以，我们的第一个任务是找到最粗粒度的语言结构，将它作为我们的起始规则。在英语中，我们可以使用sentence规则作为起始规则。对于一个XML文件，我们可以使用document规则作为起始规则。对于一个Java文件，我们可以使用compilationUnit规则作为起始规则。

设计起始规则的内容实际上就是使用“英语伪代码”来描述输入文本的整体结构，这和我们编写软件的过程有点类似。例如，“一个CSV文件

就是一系列以换行符为终止的行。”（**a comma-separated-value[CSV]file is a sequence of rows terminated by newlines.**）其中，**is a**左侧的单词**file**就是规则名，右侧的全部内容就是规则定义中的<<**stuff**>>。

```
file : «sequence of rows that are terminated by newlines» ;
```

接着，我们降低一个层次，描述起始规则右侧所指定的那些元素。它右侧的名词通常是词法符号或者尚未定义的规则。其中，词法符号是那些我们的大脑能够轻易识别出的单词、标点符号或者运算符。正如英语语句中的单词是最基本元素一样，词法符号是文法的基本元素。起始规则引用了其他的、需要进一步细化的语言结构，如上面的例子中的“行”。

再降低一层，我们可以说，一个行就是一系列由逗号分隔的字段（**a row is a sequence of fields separated by commas**）。接下来，一个字段就是一个数字或者字符串（**a field is a number or string**）。我们的伪代码如下所示：

```
file  : «sequence of rows that are terminated by newlines» ;  
row   : «sequence of fields separated by commas» ;  
field : «number or string» ;
```

当我们完成对规则的定义后，我们的语法草稿就成形了。让我们来试着用这种方法描述一下**Java**文件的关键结构（我们用斜体来突出规则名）。在最粗的粒度上，一个**Java**的编译单元（**compilation unit**）由一

个可选的包声明语句（`package specifier`）和一个或多个类定义（`class definition`）组成。其中，类定义由关键字`class`开始，之后是一个标识符、可选的父类名（`superclass specifier`）、可选的实现语句

（`implements clause`），以及类的定义体（`class body`）。一个类的定义体就是由花括号包裹的一系列类成员（`class member`）。一个类成员可以是内部类定义、字段或者方法。然后，我们将会描述字段和方法，接下来是方法中的语句。你应该已经明白了这个过程，从最高的层次开始，逐渐向下进行，将像是Java类定义这样巨大的语言结构分解为若干条稍后定义的规则。我们现在能够写出如下的语法伪代码：

```
compilationUnit : <<optional packageSpec then classDefinitions>> ;
packageSpec    : 'package' identifier ';' ;
classDefinition :
    'class' <<optional superclassSpec optional implementsClause classBody>> ;
superclassSpec : 'super' identifier ;
implementsClause :
    'implements' <<one or more identifiers separated by comma>> ;
classBody       : '{' <<zero-or-more members>> '}' ;
member          : <<nested classDefinition or field or method>> ;
...
```

如果以现有的语法规则作为参考，那么设计类似Java的大型编程语言的ANTLR语法就会容易得多。不过，盲目地遵循已经存在的语法规则可能使你误入歧途，我们接下来将会讨论这一点。

5.2 以现有的语法规则为指南

一份非**ANTLR**格式的语法规范能够很好地指导编程者理清该语言的结构。至少，一份已经存在的语法规范给我们提供了一份非常好的、可供参考的规则名列表。不过，还是小心为妙。我不建议从一门语言的参考手册里拷贝语法并粘贴到**ANTLR**中，然后调试到它正常工作为止。请把参考手册当作一份指南，而非一份代码。

出于使语法更清晰的目的，参考手册的范围通常都非常宽泛，这意味着其中的语法能够匹配很多实际上不合法的语句。或者，语法可能存在歧义，能够以多种方式匹配相同的输入文本。例如，一个语法可能指明，表达式可以调用类的构造器或者普通函数。问题是，二者都可以匹配类似**T (i)** 的输入文本。理想情况下，我们的语法中应该不存在任何歧义。

另一种极端情况是，参考手册中的语法对语法规则的约束可能过于严格。某些规则最好限制语法分析后的结果，而非限制语言整体的语法结构。例如，在12.4节中，我试图分析**W3C**的**XML**语言定义，却对它过于详细的细节感到一头雾水。**XML**语法明确指定，标签中的空白字符在哪些位置是必需的，在哪些位置是可选的。知道这一点固然很好，但是我们可以实现一个更简单的词法分析器，它只需在将数据传递给语法分析器之前丢弃标签中的全部空白字符即可。我们的语法没有必要为所有位置上的空白字符建立测试。

上述XML规范还指出，`<? xml...>`标签可以有两个特殊的属性：`encoding`和`standalone`。我们需要明白这个限制，不过，一种更加容易的实现方法是：先暂时允许任意的属性名，在完成语法分析之后，检查语法分析树来确保所有的限制都已经生效了。最后，XML仅仅是一系列嵌在文本中的标签，所以它的语法结构非常直观。唯一的挑战是如何将标签内外的内容分开对待。我们将会在第12.3节中进一步学习。

在刚开始的时候，辨识一条语法规则并使用伪代码编写右侧的内容是一项充满挑战的工作，不过，它会随着你为不同语言编写语法的过程变得越来越容易。在学习本书例子的过程中，你将会得到充分的锻炼。

一旦我们拥有了伪代码，我们就需要将它翻译为ANTLR标记，从而得到一个能够正常工作的语法。在下一节里，我们将会定义常见的四种语言模式，研究如何将它们构造成ANTLR语法。之后，我们将会学习如何定义语法中引用的词法符号，如“整数”和“标识符”。记住，在本章中，我们学习的是语法开发过程中的基础知识，这些知识将会为下一章中对真实世界的编程语言的处理奠定坚实的基础。

5.3 使用ANTLR语法识别常见的语言模式

现在，我们已经掌握了一种自顶向下的、草拟一个语法的策略，接下来我们需要关注的是常见的语言模式：序列（sequence）、选择

(choice)、词法符号依赖 (token dependency)，以及嵌套结构 (nested phrase)。在之前的章节中，我们见过这些模式的一些例子。随着学习的深入，我们会用正式的语法规则将特定的模式表达出来，通过这种方式，我们就能够掌握基本的ANTLR标记的用法。下面，让我们开始学习这些最常见的语言模式吧。

1. 序列模式

在计算机编程语言中，这种结构最常见的形式是一列元素，就像上文中的类定义中包括一系列方法一样。即使是像HTTP、POP和SMTP这样的简单的“协议语言”中，也能够看到序列模式的身影。协议的输入通常是一列指令。例如，下面是登录一台POP服务器并获取第一条消息的指令序列：

```
USER parrt  
PASS secret  
RETR 1
```

每个指令自身也是一个序列。大多数指令由一个类似USER和RETR的关键字（保留字），一个操作数和一个换行符构成。在上述例子中，我们可以说一个检索指令就是一个关键字，后面跟着一个整数，再后面是一个换行符。使用语法来表述这样的序列，我们只需要按照顺序将它们列出即可。在ANTLR标记中，检索指令可表达为：'RETR'INT'\n'，其中，INT代表整数类型的词法符号。


```
retr : 'RETR' INT '\n' ; // 匹配“关键字 – 整数 – 换行符”序列
```

注意，我们可以直接使用类似'RETR'的常量字符串来表示任意简单字符序列，诸如关键字或者标点符号等（我们将会在第5.5节中探讨类似INT的词法结构）。

我们使用语法规则来为编程语言的特定结构命名，这就好像我们在编程时将若干个语句组合成一个函数。在上例中，我们将RETR命名为retr规则。这样，在语法的其他地方，我们可以直接把规则名作为简称来引用RETR。

接下来让我们看一个任意长度序列的例子，在Matlab中，向量是保存在形如[123]的一列整数中的。对于有限长度的序列，我们可以逐个列出其中的元素，但是在这种情况下，我们不能通过INT INT INT INT INT INT INT INT INT INT...方式来列举所有可能的情况。

我们使用+字符来处理这种一个或多个元素的情况。例如，（INT）+描述一个任意长度的、整数组成的序列。作为简写，INT+也是可以的。如果这样的序列可以为空，那么我们使用代表“零个或多个元素”的*字符：INT*。上述字符好比是编程语言中的循环，当然，ANTLR自动生成的语法分析器也是通过循环来实现它们的功能的。

序列模式的变体包括：带终止符的序列模式和带分隔符的序列模式。CSV文件同时使用了这两种模式。下面是我们在先前的章节中使用

ANTLR标记写出的伪代码语法:

```
file : (row '\n')* ;           // 以一个 '\n' 作为终止符的序列
row  : field (',' field)* ;    // 以一个 ',' 作为分隔符的序列
field: INT ;                   // 假设字段都是整数
```

file规则使用带终止符的序列模式来匹配零个或多个**row**“\n”序列。其中序列中的每个元素都以“\n”字符结束。**row**规则使用带分隔符的序列模式来匹配一个**field**后面跟着零个或多个“,” **field**序列的情形。“,” 隔开了所有的**field**。**row**规则匹配类似1、1, 2以及1, 2, 3的序列。

我们再来看看其他编程语言里的相同结构, 例如, 下面的语法匹配类似Java的、每个语句都由分号结束的编程语言:

```
stats : (stat ';' )* ; // 匹配零个或多个以 ';' 终止的语句
```

与之相似, 下面的语法匹配以逗号分隔的多个表达式, 我们可以在一次函数调用的参数列表中找到这样的例子:

```
exprList : expr (',' expr)* ;
```

就连ANTLR元语言也使用了序列模式。下面的语法片段显示了ANTLR是如何使用它自身的句法表达“规则定义”这条句法的:

```
// 匹配这样的结构: ' 规则名 :' 后面跟着至少一个备选分支,
// 然后是若干条以 '|' 符号分隔的备选分支, 最后是一个 ';'
rule : ID ':' alternative ( '|' alternative )* ';' ;
```

最后，还有一种特殊的“零个或一个元素的序列”，它可以用`?` 字符来指定，用于表达一种“可选的”结构。在Java语法中，我们能够发现

`('extends'identifier)?` 这样的字符串，它用于匹配可选的父类声明。相似地，为了匹配可选的变量初始化语句，我们可以写成

`('='expr)?`。这有点像是在“有”和“无”之间选择。在下一节中我们会看到，`('='expr)?` 等价于 `('='expr|)`。

2.选择模式（多个备选分支）

如果一门编程语言只有一种语句，那就太无聊了。即使是网络协议这样的最简单的语言，也包含多种有效语句，如POP协议中的USER和RETR指令。这促使我们思考选择模式的必要性。我们已经在Java语法伪代码的member规则中看到了一个选择模式的实际应用：`<<ested class definition or field or method>>`。

我们使用`|`符号作为“或者”来表达编程语言中的选择模式，在ANTLR的规则中，它用于分隔多个可选的语法结构——称作备选分支

`(alternatives)` 或者可生成的结果 `(productions)`。选择模式在语法中随处可见。

回到之前的CSV语法，我们可以编写一条更加灵活的field规则，允许字段中出现整数或者字符串。

```
field : INT | STRING ;
```

在下一章的语法中我们可以看到很多选择模式的例子，例如6.4中的type规则里列出的许多类型名：

```
type:   'float' | 'int' | 'void' ; // 用户定义的类型
```

在6.3节中，我们在图的描述中可以看到，其中列出了所有可能出现的语句：

```
stmt:   node_stmt  
      |   edge_stmt  
      |   attr_stmt  
      |   id '=' id  
      |   subgraph  
      ;
```

任何时候，当你说到“语言结构x可以是这样或者那样”时，你就需要用到选择模式，也就是在x规则中使用|。

语法中的序列模式和选择模式使我们能够编写语言的框架，但是这还不够，接下来，还有两种关键的模式要学习：词法符号依赖和嵌套结构。在语法中，它们通常是一起出现的，不过，为简单起见，我们先来单独分析词法符号依赖模式。

3.词法符号依赖模式

在之前的例子中，我们使用INT+来表示Matlab的向量中的整数序列[1 2 3]。为了描述向量两侧的方括号，我们需要一种方法来表达对这样的词

法符号的依赖。此时，如果我们在某个语句中看到了某个符号，我们就必须在同一个语句中找到和它配对的那个符号。为表达出这种语义，在语法中，我们使用一个序列来指明所有配对的符号，通常这些符号会把其他元素分组或者包裹起来。在上例中，我们可以用下面这种方式指定一个完整的向量：

```
vector : '[' INT+ ']' ; // [1], [1 2], [1 2 3], ...
```

查看任何一个用你喜欢的编程语言编写的程序，你就会发现，几乎所有的用于分组的符号都是成对出现的：（...），{...}以及[...]。从6.4节的学习中我们能够发现，在方法调用的圆括号间，以及用于数组索引的方括号间，词法符号依赖模式都存在。

```
expr:  expr '(' exprList? ')' // 类似 f(), f(x), f(1,2) 的函数调用
      |  expr '[' expr ']'    // 类似 a[i], a[i][j] 的数组索引
      ...
      ;
```

我们也在方法声明中看到左右圆括号之间的词法符号依赖模式。

examples/Cymbol.g4

```
functionDecl
:   type ID '(' formalParameters? ')' block // "void f(int x) {...}"
;
formalParameters
:   formalParameter (',' formalParameter)*
;
formalParameter
:   type ID
```

下面这段语法来自6.2节，它匹配由一对花括号包裹的对象定义，例如 `{"name": "parrrt", "passwd": "secret"}`。

```
examples/JSON.g4
object
    :   '{' pair (',' pair)* '}'
    |   '{' '}' // 空对象
    ;
pair:   STRING ':' value ;
```

更多词法符号配对的例子，参见6.5节。

请记住，一个有依赖的符号并非必须匹配到它所依赖的符号。在C语言基础上发展起来的编程语言通常拥有 `a? b: c` 三元运算符，只有在这种情况下，`?` 才依赖其后的 `:`。

此外，词法符号间的依赖并不意味着一定存在嵌套结构。例如，一个向量中可能不允许出现嵌套的向量。不过，通常情况下，被匹配的符号包裹的内容是典型的嵌套结构。我们很容易见到类似 `a[(i)]` 和 `{while (b) {i=1; }}` 的结构。这就是我们需要学习的最后一种语言模式。

4. 嵌套模式

嵌套的词组是一种自相似的语言结构，即它的子词组也遵循相同的结构。表达式是一种典型的自相似语言结构，它包含多个嵌套的、以运算符分隔的子表达式。与之相似，一个 `while` 循环代码块是一个嵌套在

更外层代码块中的代码块。在语法中，我们使用递归规则来表达这种自相似的语言结构。所以，如果一条规则定义中的伪代码引用了它自身，我们就需要一条递归规则（自引用规则）。

让我们看一看如何处理“代码块”这样的嵌套结构。一个**while**表达式由一个关键字**while**开始，后面是一个在括号中的条件表达式，再后面就是一条语句。我们也可以把多个语句放入花括号中，当作一个“代码块语句”使用。对上述规则的语法表述如下所示：

```
stat: 'while' '(' expr ')' stat // 匹配 WHILE 语句
    | '{' stat* '}'           // 匹配花括号中若干条语句组成的代码块
    ...                       // 其他种类的语句
    ;
```

其中，**while**中的**stat**是一个循环结构，它可以是一个语句或者由花括号包裹的一组语句。因为**stat**规则在前两个备选分支中引用了自身，我们称它为直接递归（**directly recursive**）的。如果我们将它的第二个备选分支抽取出来，**stat**规则和**block**规则就会互为间接递归（**indirectly recursive**）的。

```
stat: 'while' '(' expr ')' stat // 匹配 WHILE 语句
    | block                    // 匹配一个语句组成的代码块
    ...                       // 其他种类的语句
    ;
block: '{' stat* '}' ;         // 匹配花括号中若干条语句组成的代码块
```

大部分编程语言都包含多种形式的自相似结构，这带来的结果是语法中包含很多递归规则。让我们一起来看一门简单的、表达式类型只有

三种——数组索引表达式、括号表达式和整数——的编程语言。下面是用ANTLR标记书写的语法：

```
expr: ID '[' expr ']' // a[1], a[b[1]], a[(2*b[1])]
     | '(' expr ')'    // (1), (a[1]), (((1))), (2*a[1])
     | INT             // 1, 94117
     ;
```

其中的递归发生的非常自然。因为一个数组的索引值本身也是一个表达式，所以我们就在对应的备选分支中直接引用了**expr**。实际上，索引值本身也可以是一个数组索引表达式。从这个例子中我们可以看到，语言结构上的递归自然而然地使得语言规则发生了递归。如图5-1所示是两个样例输入对应的语法分析树。

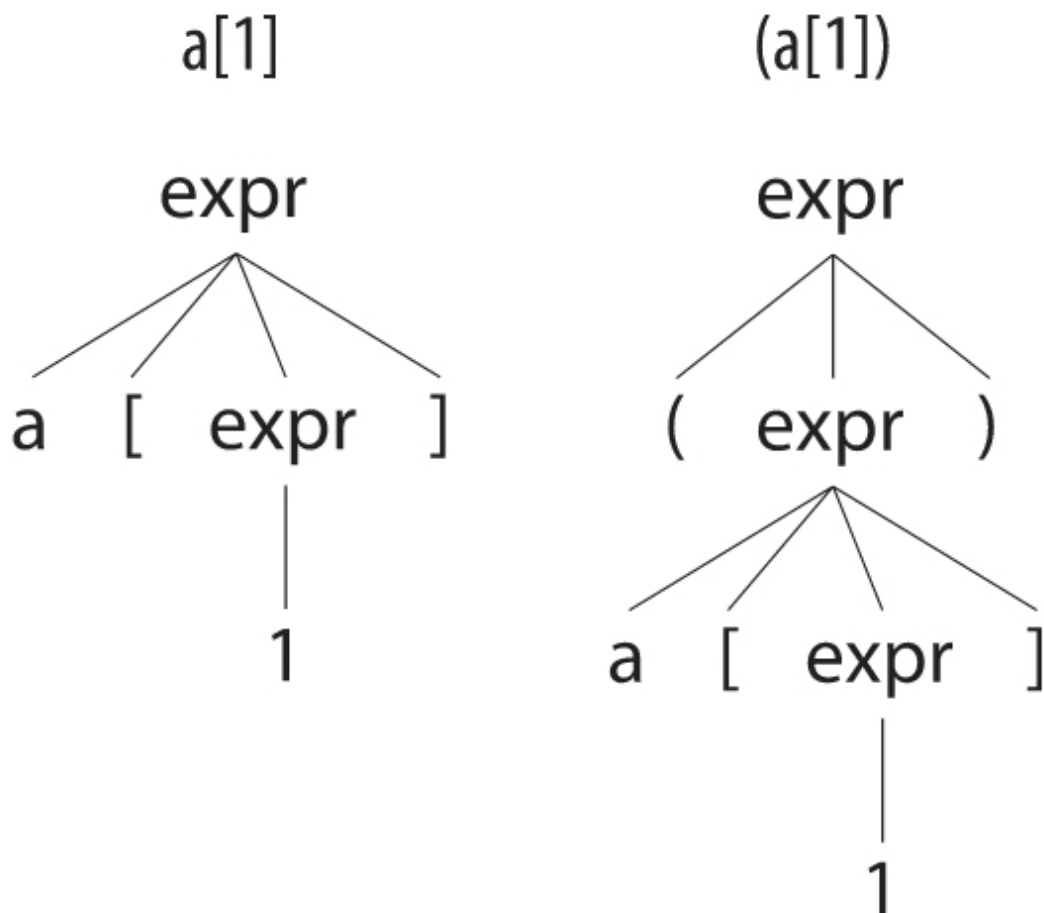


图5-1 两个样例输入对应的语法分析树

正如我们在2.1节中看到的那样，语法分析树的非叶子节点代表了规则，而叶子节点代表了词法符号。一条从根节点到任意节点的路径代表了对应的规则调用栈（同时也是ANTLR自动生成的递归下降语法分析器的调用栈）。因此，代表递归调用的路径上就会存在对多个相同规则的引用。我喜欢将一个规则节点看作它的后代子树的标签：因为根节点是`expr`，所以整棵树就是一个表达式（`expression`）。在上面的例子中，1的父节点`expr`说明，整数1是一个表达式。

并非所有的语言都有表达式，例如数据格式定义。不过，你所接触的大多数语言都包含非常复杂的表达式（参见6.5节）。此外，有些语法规则中对表达式的描述并不是十分清楚，所以稍后我们会花些时间来深入研究识别表达式的细节，这是一件很有价值的事情。

表5-1总结了ANTLR的核心语法标记，以备后续章节引用。

表5-1 ANTLR核心标记

用 法	描 述
x	匹配词法符号、规则引用或者子规则 x
x y ... z	匹配一系列规则元素
(... )	一个具有多个备选分支的子规则
x?	匹配 x 或者忽略它
x*	匹配 x 零次或多次
x +	匹配 x 一次或多次
r : ... ;	定义规则 r
r : ;	定义具有多个备选分支的规则 r

迄今为止，我们学习了一些常见的计算机语言的模式，对它们的总结见表5-2。

表5-2 几种常见的计算机语言的模式

模式名	描 述
序列模式	<p>它是一个有限长度或者任意长度的序列，序列中的元素可以是词法符号或者子规则。序列模式的例子包括变量声明（类型后面紧跟着标识符）和整数序列，下面是范例实现：</p> <pre> x y ... z // x 后面跟着 y, ..., z {' INT+ '}' // Matlab 的整数向量 </pre>
带终止符的序列模式	<p>它是一个任意长的、可能为空的序列，该序列由一个词法符号分隔开，通常是分号或者换行符，其中的元素可以是词法符号或者子规则。这样的例子包括类 C 语言的语句集合和一些用换行符来分隔的数据格式。下面是范例实现：</p> <pre> (statement ';')* // Java 的语句集合 (row '\n')* // 多行数据 </pre>
带分隔符的序列模式	<p>它是一个任意长的、可能为空的序列，该序列由一个词法符号分隔开，通常是逗号、分号或是句号，其中的元素可以是词法符号或者子规则。这样的例子包括函数定义中的参数表、函数调用时传递的参数表、某些语句之间有分隔符却无终止符的编程语言^②，以及目录名。下面是范例实现：</p> <pre> expr (',' expr)* // 函数调用时传递的参数 (expr (',' expr)*)? // 函数调用时传递的参数是可选的 '/'? name ('/' name)* // 简化的目录名 stat ('.' stat)* // 若干个 SmallTalk 语句 </pre>
选择模式	<p>它是一组备选分支的集合。这样的例子包括不同类型的类型、语句、表达式或者 XML 标签。下面是范例实现：</p> <pre> type : 'int' 'float' ; stat : ifstat whilestat 'return' expr ';' ; expr : '(' expr ')' INT ID ; tag : '<' Name attribute* '>' '<' '/' Name '>' ; </pre>
词法符号依赖	<p>一个词法符号需要和一个或者多个后续词法符号匹配。这样的例子包括配对的圆括号、花括号、方括号和尖括号。下面是范例实现：</p> <pre> '(' expr ')' // 嵌套表达式 ID '[' expr ']' // 数组索引表达式 {' stat* '}' // 花括号包裹的若干个语句 '<' ID (',' ID)* '>' // 泛型声明 </pre>
嵌套模式	<p>它是一种自相似的语言结构。这样的例子包括表达式、Java 的内部类、嵌套的代码块以及嵌套的 Python 函数定义。下面是范例实现：</p> <pre> expr : '(' expr ')' ID ; classDef : 'class' ID '{' (classDef method field) '}' ; </pre>

（注：SmallTalk 语言的语句之间用.分隔，最后一条语句后不需要。
——译者注）

5.4 处理优先级、左递归和结合性

在自顶向下的语法和手工编写的递归下降语法分析器中，处理表达式都是一件相当棘手的事情，这首先是因为大多数语法都存在歧义，其次是因为大多数语言的规范使用了一种特殊的递归方式，称为左递归（left recursion）。我们稍后会详细讨论它，现在请记住一点，自顶向下的语法和语法分析器的经典形式无法处理左递归。为了阐明这个问题，假设有一种简单的算术表达式语言，它包含乘法和加法运算符，以及整数因子。表达式是自相似的，所以，很自然地，我们说，一个乘法表达式是由*连接的两个子表达式，一个加法表达式是由+连接的两个子表达式。另外单个整数也可以作为简单的表达式。这样写出的就是下列看上去非常合理的规则：

```
expr : expr '*' expr // 匹配由 '*' 运算符连接的子表达式
      | expr '+' expr // 匹配由 '+' 运算符连接的子表达式
      | INT           // 匹配简单的整数因子
      ;
```

问题在于，对于某些输入文本而言，上面的规则存在歧义。换句话说，这条规则可以用不止一种方式匹配某种输入的字符流，正如2.3节中所描述的那样。这个语法在简单的整数表达式和单运算符表达式上工作得很好——例如1+2和1*2——是因为只存在一种方式去匹配它们。对于1+2，上述语法只能用第二个备选分支去匹配，如图5-2左侧的语法分析树所示。

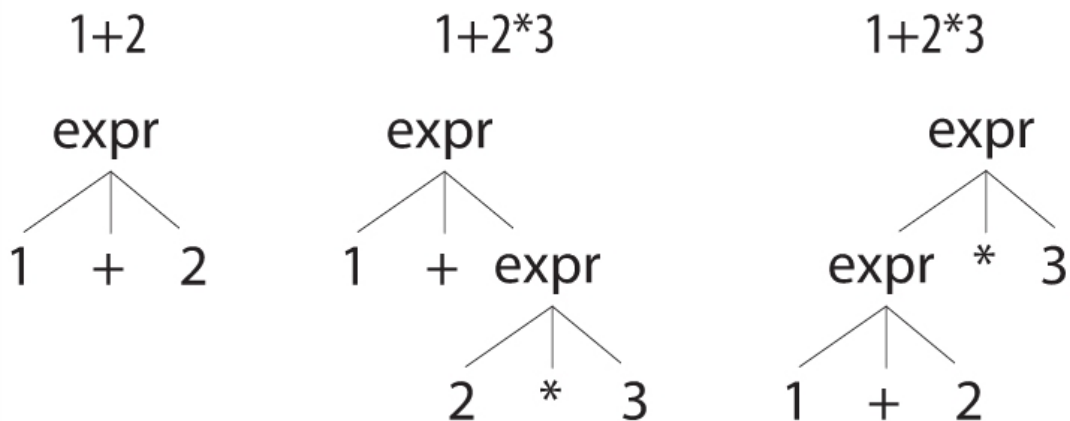


图5-2 按照不同方式解释的语法分析树

但是对于 $1+2*3$ 这样的输入而言，上述规则能够用两种方式解释它，如图5-2中间和右侧的语法分析树所示。它们的差异在于，中间的语法分析树表示将1加到2和3相乘的结果上去，而右侧的语法分析树表示将1和2相加的结果与3相乘。

这就是运算符优先级带来的问题，传统的语法无法指定优先级。大多数语法工具，例如Bison，使用额外的标记来指定运算符优先级。

与之不同的是，ANTLR通过优先选择位置靠前的备选分支来解决歧义问题，这隐式地允许我们指定运算符优先级。例如，`expr`规则中，乘法规则在加法规则之前，所以ANTLR在解决 $1+2*3$ 的歧义问题时会优先处理乘法。默认情况下，ANTLR按照我们通常对*和+的理解，将运算符从左向右地进行结合。尽管如此，一些运算符——例如指数运算符——是从右向左结合的，所以我们需要在这样的运算符上使用`assoc`选

项手工指定结合性。这样，输入的 2^3^4 就能够被正确解释为 $2^{(3^4)}$ ：

```
expr : expr '^'<assoc=right> expr // ^ 运算符是右结合的
      | INT
      ;
```

注：在ANTLR 4.2之后，<assoc=right>需要被放到备选分支的最左侧，否则会收到警告。在本例中，正确写法是：

```
expr : <assoc=right> expr '^'expr
      | INT
      ;
```

如图5-3所示的语法分析树展示了 \wedge 符号的左结合版本和右结合版本在处理相同输入时的差异。通常人们采用右侧语法分析树所代表的解释方式，不过，语言设计者可以自由地决定使用哪一种结合性。

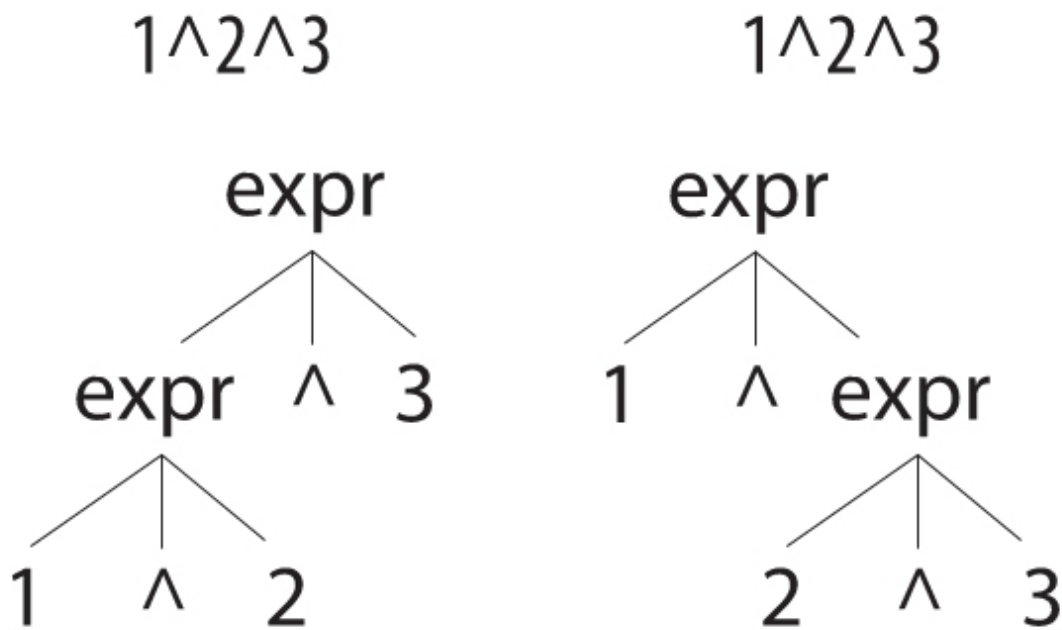


图5-3 展示了处理相同输入时差异的语法分析树

若要将上述三种运算符组合成为同一条规则，我们就必须把 \wedge 放在最前面，因为它的优先级比 $*$ 和 $+$ 都要高（ $1+2\wedge 3$ 的结果是9）。

```

expr : expr '^'<assoc=right> expr // ^ 运算符是右结合的
      | expr '*' expr // 匹配由 '*' 运算符连接的子表达式
      | expr '+' expr // 匹配由 '+' 运算符连接的子表达式
      | INT // 匹配简单的整数因子
      ;

```

熟悉ANTLR 3的读者可能正在等我指出，和所有传统的自顶向下的语法分析器生成器一样，ANTLR无法处理左递归规则。然而，ANTLR 4的一项重大改进就是，它已经可以处理直接左递归了。左递归规则是这样的一种规则：在某个备选分支的最左侧以直接或者间接方式调用了自身。上面的例子中的`expr`规则是直接左递归的，因为除`INT`之外的

所有备选分支都以`expr`规则本身开头（它同时也是右递归（`right recursive`）的，因为它的某些备选分支在最右侧引用了`expr`）。

虽然ANTLR 4已经能够处理直接左递归，但是它还无法处理间接左递归。这意味着我们无法将`expr`规则分解为下列规则，尽管它们在语义上等价：

```
expr : expo // 通过 expo 规则间接左递归地调用了 expr 规则
      | ...
      ;
expo : expr '^'<assoc=right> expr ;
```

使用优先级上升（**Precedence Climbing**）算法解析表达式

经验丰富的编译器作者通常会手工编写递归下降的语法分析器，以此榨干计算机的最后一滴性能，同时完全地掌控从错误中恢复的过程。通常，它们不编写一长串表达式规则，而是使用带运算符优先级的语法分析器。

ANTLR使用的机制和运算符优先级相似，但是更加强大，它主要源于Keith Clarke自1986年起的工作。Theodore Norvell随后创造了术语优先级上升（`precedence climbing`）。相似地，ANTLR将直接左递归替换为一个判定循环（`predicated loop`），该循环会比较前一个和下一个运算符的优先级。我们将在第11章中深入了解判定。

在ANTLR 3中，为了识别表达式，我们必须将之前出现过的、左递归的expr规则拆分成多条规则，每个优先级对应一条规则。例如，我们使用下面的规则来处理带加号和乘号的表达式。

```
expr      : addExpr ;
addExpr   : multExpr ('+' multExpr)* ;
multExpr  : atom ('*' atom)* ;
atom      : INT ;
```

像C和Java这样的语言中，最终描述表达式的规则大概有十五条，不论是构造一个自顶向下的语法，还是手工构建一个递归下降的语法分析器，这样复杂的规则都带来了巨大的工作量。

ANTLR 4简化了处理（直接）左递归表达式规则的相关工作。新的机制不仅更有效率，而且使表达式规则更简洁、更易理解。例如，在Java语法中，用于描述表达式的规则的行数下降了一半（从172行下降到91行）。

在实践中，我们可以用直接左递归来处理所有我们关注的语言结构。例如，下面的规则匹配C语言中的部分声明语句，如* (*a) []。

```
decl : decl '[' ']' // 使用直接左递归匹配 [] 后缀
      | '*' decl    // *x, *x[], **x
      | '(' decl ')' // (x), (x[]), (*x)[]
      | ID
      ;
```

欲了解更多ANTLR（使用语法变换）支持左递归的细节，请参阅第14章。

迄今为止，我们已经学习了计算机语言中的常见模式，也懂得了如何使用ANTLR标记来表达这些模式。不过，在深入研究完整的范例程序之前，我们需要搞清楚如何在语法规则中描述其引用的词法符号。正如存在若干种关键的语法模式一样，我们发现，编程语言中还存在着一些极其常见的词法结构。编写一个完整的语法实际上就是将本节讲述的语法规则和下一节讲述的词法规则组合在一起的过程。

5.5 识别常见的词法结构

在词法角度上，不同的计算机语言的外观都十分相似。例如，如果我打乱一段输入文本的顺序，然后分别在所有曾经出现过的编程语言中将词法符号）10（f重新组合为有效的词组，会发生什么呢？五十年前，我们在LISP中看到是（f 10），在Algol中看到是f（10）。实际上，f（10）在从Prolog到Java再到Go语言的几乎所有编程语言中都是有效的。在词法角度上，不论是函数式、过程式、声明式，还是面向对象的编程语言，看上去都是大同小异的。这一点令人惊讶。

这是一件好事，因为我们只需描述标识符和整数一次，然后稍加改动，就可以将它们应用于大多数的编程语言中。和语法分析器一样，词法分析器也使用规则来描述种类繁多的语言结构。在ANTLR中，我

们使用的是几乎完全相同的标记。唯一的差别在于，语法分析器通过输入的词法符号流来识别特定的语言结构，而词法分析器通过输入的字符流来识别特定的语言结构。

由于词法规则和文法规则的结构相似，**ANTLR**允许二者在同一个语法文件中同时存在。不过，由于词法分析和语法分析是语言识别过程中的两个不同阶段，我们必须告诉**ANTLR**每条规则对应的阶段。它是通过这种方式完成的：词法规则以大写字母开头，而文法规则以小写字母开头。例如，**ID**是一个词法规则名，而**expr**是一个文法规则名。

当开始编写一个新语法的时候，我通常从一个已有的语法（例如**Java**语法）中复制一些常见的词法结构对应的规则：标识符、数字、字符串、注释，以及空白字符。几乎所有的语言，哪怕是**XML**和**JSON**这样的非编程类的语言，都包含这些词法符号的变体。例如，尽管二者的语法差异巨大，**C**语言的词法分析器还是能够毫无问题地对下面的**JSON**字符流进行词法分析。

```
{
  "title": "Cat wrestling",
  "chapters": [ {"Intro": "..."}, ... ]
}
```

另外一个例子是多行注释。在**Java**中，多行注释使用`/*...*/`，而在**XML**中，多行注释使用的是`<!--...-->`，除了开始和结束的字符不同之外，

二者的词法结构几乎完全相同。

对于关键字、运算符和标点符号，我们无须声明词法规则，只需要在语法规则中直接使用单引号将它们括起来即可，例如'**while**'、'*'，以及'++'。有些开发者更愿意使用类似**MULT**的词法规则来引用'*'，以避免对其的直接使用。这样，在改变乘法运算符的时候，它们只需修改**MULT**规则，而无须逐个修改引用了**MULT**的语法规则。

为了展示词法规则，让我们一起来构造一些描述常见词法符号的词法规则的简化版本，下面就从我们的老朋友标识符开始吧。

1. 匹配标识符

在语法的伪代码中，一个基本的标识符就是一个由大小写字母组成的字符序列。我们知道，可以使用刚刚掌握的方法 (...) + 来表达序列模式。因为序列中的元素既可以是写字母也可以是小写字母，我们还知道，应当在子规则中使用选择运算符：

```
ID : ('a'..'z'|'A'..'Z')+ ; // 匹配 1 个或多个大小写字母
```

上面的**ANTLR**标记中，唯一让我们感到新鲜的是范围运算符：'a'..'z'，它的意思是从a到z的所有字符。这实际上是从97到122的**ASCII**码。如果我们需要使用**Unicode**字符（**Unicode code point**），就必须写

作'\uXXXX'，其中XXXX是相应的Unicode字符以十六进制表示的码点值。此外，ANTLR还支持正则表达式中用于表示字符集的缩写：

```
ID : [a-zA-Z]+ ; // 匹配 1 个或多个大小写字母
```

类似ID的规则有时候会和其他词法规则或者字符串常量值产生冲突，例如'enum'。

```
grammar KeywordTest;
enumDef : 'enum' '{' ... '}' ;
...
FOR : 'for' ;
...
ID : [a-zA-Z]+ ; // 不会匹配 'enum' 和 'for'
```

ID规则也能够匹配类似enum和for的关键字，这意味着存在不止一种规则可以匹配相同的输入字符串。要弄清此事，我们需要了解ANTLR对这种混合了词法规则和文法规则的语法文件的处理机制。首先，ANTLR从文法规则中筛选出所有的字符串常量，并将它们和词法规则放在一起。'enum'这样的字符串常量被隐式定义为词法规则，然后放置在文法规则之后、显式定义的词法规则之前。ANTLR词法分析器解决歧义问题的方法是优先使用位置靠前的词法规则。这意味着，ID规则必须定义在所有的关键字规则之后，在上面的例子中，它在FOR规则之后。ANTLR将为字符串常量隐式生成的词法规则放在显式定义的词法规则之前，所以它们总是拥有最高的优先级。因此，在本例中，'enum'被自动赋予了比ID更高的优先级。

因为ANTLR自动将词法规则放置在文法规则之后，下面的KeywordTest语法的变体会生成相同的语法分析器和词法分析器：

```
grammar KeywordTestReordered;
FOR : 'for' ;
ID  : [a-zA-Z]+ ; // 不会匹配 'enum' 和 'for'
...
enumDef : 'enum' '{' ... '}' ;
...
```

上述标识符中不允许出现数字，不过你可以预习6.3节至6.5节的内容来了解ID规则的完整定义。

2. 匹配数字

描述10这样的数字非常容易，它不过是一列数字而已。

```
INT : '0'..'9'+ ; // 匹配 1 个或多个数字
```

或者：

```
INT : [0-9]+ ; // 匹配 1 个或多个数字
```

不幸的是，浮点数要复杂得多，不过，我们可以先完成一个简化的版本，忽略掉指数形式（关于完整的匹配浮点数的词法规则定义，可参阅6.5节，其中的规则甚至可以匹配类似3.2i的复数）。一个浮点数以一系列数字为开头，后面跟着一个点，然后是可选的小数部分；浮点数的

另外一种格式是，以点为开头，后面是一列数字。一个单独的点不是一个合法的浮点数定义。基于上述格式，我们的浮点数规则使用了选择模式和序列模式。

```

FLOAT:  DIGIT+ '.' DIGIT*  // 匹配 1. 39. 3.14159 等 ...
        |
        | '.' DIGIT+  // 匹配 .1 .14159
        ;

```

fragment

```

DIGIT   :   [0-9] ;           // 匹配单个数字

```

在这里，我们使用了一条辅助规则**DIGIT**，这样就不用重复书写**[0-9]**了。将一条规则声明为**fragment**可以告诉**ANTLR**，该规则本身不是一个词法符号，它只会被其他的词法规则使用。这意味着我们不能在文法规则中引用**DIGIT**。

3.匹配字符串常量

另外一种计算机语言共有的词法符号是类似**"Hello"**的字符串常量。大多数语言中的字符串常量使用双引号，部分语言使用单引号或者同时使用单引号和双引号（**Python**）。不论哪种分界符，我们都使用同一种规则来匹配字符串常量：识别分界符之间的全部内容。

用语法伪代码表示，一个字符串就是两个双引号之间的任意字符序列。

```

STRING :  '"' .*? '"' ; // 匹配 "... " 间的任意文本

```

其中，点号通配符匹配任意的单个字符。因此，`.*`就是一个循环，它匹配零个或多个字符组成的任意字符序列。显然，它可以一直匹配到文件结束，但这没有任何意义。为解决这个问题，**ANTLR**通过标准正则表达式的标记（`?` 后缀）提供了对非贪婪匹配子规则（**nongreedy subrule**）的支持。非贪婪匹配的基本含义是：“获取一些字符，直到发现匹配后续子规则的字符为止”。更准确的描述是，在保证整个父规则完成匹配的前提下，非贪婪的子规则匹配数量最少的字符。有关非贪婪匹配的更多细节，请参阅15.6节。与之相反，`.*`是贪婪的，因为它贪婪地消费掉一切匹配的字符（在本例中就是匹配通配符`.`的字符）。如果`.*?`令你感到迷惑不解，不要担心，只需要记住它是一种匹配双引号或者其他分界符之间的东西的模式即可。不久之后，我们在研究注释的章节中还会见到非贪婪循环。

我们的**STRING**规则还不够完善，因为它不允许其中出现双引号。为了解决这个问题，很多语言都定义了以`\`开头的转义序列。在这些语言中，如果希望在一个被双引号包围的字符串中使用双引号，我们就需要使用`\`。下列规则能够支持常见的转义字符：

```
STRING: '"' (ESC|.)*? '"' ;  
fragment  
ESC : '\\"' | '\\\\' ; // 双字符序列 \" 和 \\
```

其中，**ANTLR**语法本身需要对转义字符`\`进行转义，因此我们需要`\\`来表示单个反斜杠字符。

现在，**STRING**规则中的循环既能通过**ESC**片段规则（**fragment rule**）来匹配转义字符序列，也能通过通配符来匹配任意的单个字符。***?** 运算符会使（**ESC|.**）***?** 循环在看到后续子规则，即一个未转义的双引号时终止。

4.匹配注释和空白字符

当词法分析器匹配到我们刚刚定义过的那些词法符号的时候，它会将匹配到的词法符号放入词法符号流，输送给语法分析器。之后，由语法分析器来检查词法符号流的语法结构。但是，当词法分析器匹配到注释和空白字符的时候，我们通常希望将它们丢弃。这样，语法分析器就不必处理注释和空白字符了。否则，下列文法规则就变成了一团乱麻，且十分容易出错，其中，**WS**是代表空白字符的词法规则：

```
assign : ID (WS|COMMENT)? '=' (WS|COMMENT)? expr (WS|COMMENT)? ;
```

定义需要被丢弃的词法符号的方法和定义正常的词法符号的方法一样。我们只需要使用**skip**指令通知词法分析器将它们丢弃即可。例如，下面是匹配类C语言中的单行和多行注释的方法：

```
LINE_COMMENT : '//' .*? '\r'? '\n' -> skip ; // 匹配 "//" 任意字符序列 '\n'
COMMENT      : '/*' .*? '*/'      -> skip ; // 匹配 "/*" 任意字符序列 "*/"
```

在**LINE_COMMENT**规则中，***?** 会消费掉//后面的一切字符，直至遇到换行符**\n**为止。（可以将本条规则放在空白字符串之前来匹配

Windows风格的换行符`\r\n`)。在COMMENT规则中，`.*?` 消费`/*`和`*/`之间的一切字符。词法分析器可以接受许多种位于`->`操作符之后的指令，`skip`只是其中之一。例如，我们能够使用`channel`指令将某些词法符号放入一个“隐藏的通道”并输送给语法分析器。更多有关词法符号通道的内容，请参阅12.1节。

现在，让我们来处理最后一种词法符号——空白字符。大多数编程语言将空白字符看作词法符号间的分隔符，并将它们忽略（Python是一个例外，它使用空白字符来达到某些语法上的目的：换行符代表一条命令的终止，特定数量的缩进指明嵌套的层级）。下列规则告诉ANTLR丢弃空白字符：

```
WS : (' '|' '\t' | '\r' | '\n')+ -> skip ; // 匹配一个或多个空白字符并将它们丢弃
```

或者：

```
WS : [ \t\r\n]+ -> skip ; // 匹配一个或多个空白字符并将它们丢弃
```

当换行符既是可忽略的空白字符，又是命令终止符的时候，我们的麻烦就来了。换行符变成了上下文相关（**context-sensitive**）的。在某种语法上下文中，我们应该丢弃换行符，在另外一些上下文中，我们需要将它输送给语法分析器，从而让语法分析器得知命令被终止了。例如，在Python中，`f()` 后面的换行符开始代码的执行，即调用函数`f`

()。但是我们也可以在括号之间插入一个额外的换行符。Python直到)后的换行符才执行函数的调用。

```
⇒ $ python
⇒ >>> def f(): print "hi"
  < ...
⇒ >>> f()
  < hi
⇒ >>> f(
⇒ ... )
  < hi
```

有关这个问题的详细解决方案，请参阅12.2节中“有趣的Python换行符”部分。

现在，我们知道了如何匹配最常见的词法结构——标识符、数字、字符串、注释以及空白字符的基础版本。信不信由你，即使是一门大型编程语言的词法分析器，也需要这些词法结构作为基础。如表5-3所示有一些基础的词法规则供我们使用，有关它们的细节将在稍后提及。

表5-3 一些基础的语法规则

词法符号类型	描述及范例
标点符号	<p>处理运算符和标点符号最容易的方式就是直接在文法规则中引用它们。</p> <pre>call : ID '(' exprList ')'</pre> <p>一些开发者更愿意定义类似 LP（左括号，left parenthesis）的词法符号标签。</p> <pre>call : ID LP exprList RP ; LP : '(' ; RP : ')'</pre>
关键字	<p>关键字是保留的标识符，我们既可以直接引用它们，也可以为它们定义词法符号类型。</p> <pre>returnStat : 'return' expr ;'</pre>
标识符	<p>几乎每种语言中的标识符看上去都差不多，它们之间的差异通常在于第一个字符的可选值以及是否允许 Unicode 字符。</p> <pre>ID : ID_LETTER (ID_LETTER DIGIT)* ; // C 语言的语法片段 fragment ID_LETTER : 'a'..'z' 'A'..'Z' '_' ; fragment DIGIT : '0'..'9' ;</pre>

（续）

词法符号类型	描述及范例
数字	<p>下列规则定义了整数和简单的浮点数。</p> <pre>INT : DIGIT+ ; FLOAT : DIGIT+ '.' DIGIT* '.' DIGIT+ ;</pre>
字符串	<p>匹配双引号包围的字符串</p> <pre>STRING : '"' (ESC .)?* '"' ; fragment ESC : '\\ ' [b t n r '\\'] ; // \b, \t, \n 等 ...</pre>
注释	<p>匹配并丢弃注释</p> <pre>LINE_COMMENT : '//' .*? '\n' -> skip ; COMMENT : '/*' .*? '*/' -> skip ;</pre>
空白字符	<p>在词法分析器中匹配空白字符并丢弃之</p> <pre>WS : [\t\n\r]+ -> skip ;</pre>

至此，我们已经知道了如何以一份样例输入文件为蓝本构造出文法规则和词法规则，这就为下一章的实战做好了准备。在继续学习之前，还需要记住重要的两点。第一，在词法规则和文法规则之间并不总是存在清晰的界线。第二，我们应当知道，ANTLR对语法规则施加了一定的限制。

5.6 划定词法分析器和语法分析器的界线

由于ANTLR的词法规则可以包含递归，从技术角度看，词法分析器变得和语法分析器一样强大。这意味着我们可以甚至可以在词法分析器中匹配语法结构。或者，另外一种极端是，我们可以把字符看作词法符号，然后用语法分析器来分析字符流的语法结构（这种情况称为无扫描器的语法分析器（`scannerless parser`），参阅 `code/extras/CSQL.g4`，匹配一门小型的C和SQL的混合语言的语法）。

划定词法分析器和语法分析器的界线位置不仅是语言的职责，更是语言编写的应用程序的职责。幸运的是，我们可以得到一些经验法则的指导。

- 在词法分析器中匹配并丢弃任何语法分析器无须知晓的东西。对于编程语言来说，要识别并丢弃的就是类似注释和空白字符的东西。否则，语法分析器就需要频繁检查它们是否存在于词法符号之间。
- 由词法分析器来匹配类似标识符、关键字、字符串和数字的常见词法符号。语法分析器的层级更高，所以我们不应当让它处理将数字组合成整数这样的事情，这会加重它的负担。
- 将语法分析器无须区分的词法结构归为同一个词法符号类型。例如，如果我们的程序对待整数和浮点数的方式是一致的，那就把它们都归为**NUMBER**类型的词法符号。没必要传给语法分析器不同的类型。

·将任何语法分析器可以以相同方式处理的实体归为一类。例如，如果语法分析器不关心XML标签的内容，词法分析器就可以将尖括号中的所有内容归为一个名为TAG的词法符号类型。

·另一方面，如果语法分析器需要把一种类型的文本拆开处理，那么词法分析器就应该将它的各组成部分作为独立的词法符号输送给语法分析器。例如，如果语法分析器需要处理IP地址中的元素，那么词法分析器就应该把IP地址的各组成部分（整数和点）作为独立的词法符号送入语法分析器。

当我们说语法分析器无须区分特定的词法结构或者无须关心某个词法结构的内容时，实际上的意思是我们编写的程序不关心它们。我们编写的程序对这些词法结构进行的处理和翻译工作与语法分析器相同。

为了展示最终的程序对我们构建词法分析器和语法分析器过程的影响，想象一个场景，我们在处理一个网络服务器上的日志文件，日志文件的每行包含一条记录。我们将逐渐增加程序的需求，在这个过程中分析词法分析器和语法分析器之间的界线是如何移动的。首先，假设每行都有一个IP地址、一个HTTP的请求方法，以及一个HTTP的状态码，下面是简单的示例：

```
192.168.209.85 "GET /download/foo.html HTTP/1.0" 200
```

我们的大脑很自然地这些不同的词法元素中提取出了信息，不过，如果我们想要的只是统计文件的总行数，我们就可以忽略除换行符之外的一切字符。

```
file  : NL+ ;           // 匹配换行符序列的语法分析器
STUFF : ~'\n'+ -> skip ; // 除 '\n' 之外的字符全部丢弃
NL    : '\n' ;         // 将设定的换行符返回给语法分析器或者其他的调用者
```

在上面的结构中，词法分析器不需要识别太多东西，语法分析器也只需匹配换行符序列（`~x`运算符匹配除`x`之外的任何字符）。

接下来，我们增加一个需求：从日志文件中提取IP地址的列表。这意味着我们需要一条匹配IP地址的词法规则，最好还有一些词法规则来匹配一行记录中的其他元素。

```
IP    : INT '.' INT '.' INT '.' INT ; // 192.168.209.85
INT   : [0-9]+ ;                      // 匹配 IP 地址中的一个字节或者 HTTP 的状态码
STRING: '"' .*? '"' ;                 // 匹配 HTTP 请求的首行
NL    : '\n' ;                       // 匹配一行记录的终止符
WS    : ' ' -> skip ;                 // 忽略空格
```

使用上面这些完整的词法符号，我们就可以构造匹配日志文件中全部记录的文法规则了。

```
file  : row+ ;           // 匹配日志文件中的全部行的文法规则
row   : IP STRING INT NL ; // 匹配日志文件中的一行记录
```

在程序的后续处理中，我们需要将文本形式的IP地址转换成为一个32位的整数。虽然我们可以将整个IP地址传给语法分析器，令其使用类似split（'.'）的方法完成处理工作，但是更好的做法是，令词法分析器匹配IP地址这种词法结构，然后将IP地址中的每个组成部分当作单独的词法符号传递给语法分析器。

```
file  : row+ ;           // 匹配日志文件中的全部行的文法规则
row   : ip STRING INT NL ; // 匹配日志文件中的一行记录
ip    : INT '.' INT '.' INT '.' INT ; // 在语法分析器中匹配 IP 地址

INT   : [0-9]+ ;         // 匹配 IP 地址中的一个字节或者 HTTP 的状态码
STRING: '"' .*? '"' ;    // 匹配 HTTP 请求的首行
NL    : '\n' ;           // 匹配一行记录的终止符
WS    : ' ' -> skip ;    // 忽略空格
```

从词法规则IP转换到文法规则ip的过程显示了，移动词法分析器和语法分析器之间的分界线这件事情有多么容易（将四个INT词法符号转换为一个32位整数需要一些内嵌在语法中的程序代码，我们还没有深入探讨过这种机制，所以暂时将其搁置）。

如果需求是处理其中的HTTP请求首行的内容，我们的思维过程与之相似。若程序无须理解HTTP请求首行中各部分的内容，词法分析器就可以将整个字符串当作一个词法符号传给语法分析器。但是，若我们的程序需要取出其中的某个部分，那么最好先让词法分析器识别出这些部分，然后将它们输送给语法分析器。

用不了多久，你就能自如地根据语言规范和程序的需求来划分这条界线了。下一章的例子将会帮助你牢记本节中的经验法则。之后，有了这样的坚实基础，我们就能在第12章中处理一些棘手的问题了。例如，Java编译器需要在忽略Javadoc注释的同时处理它；在XML文件中，标签内外的词法结构不一致。

在本章中，我们学习了如何根据一份语言的样例代码或者文档，来构造语法的伪代码，然后使用ANTLR标记构造出一个正式的语法。我们也学到了通用的语言模式：序列、选择、词法符号依赖和嵌套结构。在词法分析领域中，我们了解了最常见的词法符号的实现方法：标识符、数字、字符串、注释，以及空白字符。现在，是时候将这些知识应用于实践了，我们将会尝试构造一些真实世界中语言的语法。

第6章 探索真实的语法世界

在上一章中，我们学习了通用的词法结构和语法结构，知道了如何使用ANTLR语法来表达它们。现在，是时候使用这些知识来构造真实世界的语法了。在本章中，我们的主要目标是学习如何通过详细阅读参考手册、样例代码和已有的非ANTLR语法来构造完整的语法。我们将会循序渐进地处理五种语言。就现在而言，你不需要亲力亲为地将它们全部构造一遍，只完成你熟悉的那些即可。在未来的实践中遇到复

杂问题时，欢迎随时回来查阅本章。除此之外，你也可以随时回顾上一章中的语法模式和ANTLR语法片段。

我们要处理的第一种语言是电子表格程序和数据库经常使用的逗号分隔符（**comma-separated-value**，**CSV**）文件格式。**CSV**是一个很好的起点，因为它简单而又被广泛使用。第二种语言也是一种数据格式，称为**JSON**，它包含嵌套的数据元素，从而能够让我们通过一门真正的语言来探索递归规则的应用。

接下来，我们要研究一门名为**DOT**的声明式语言，这种语言用于描述图形（网络）。在声明式语言中，我们并非通过指定控制流来表达逻辑结构。**DOT**能让我们探索更加复杂的词法结构，例如不区分大小写的关键字。

我们研究的第四门语言是一门简单的非面向对象的编程语言**Cymbol**（在参考文献【**Language Implementation Patterns**[Par09]】的第6章也有讨论）。这是一种基于原型的语言，我们可以将它作为其他的命令式编程语言（包含函数、变量、语句和表达式）的参考或者入门。

最后，我们会为**R**函数式编程语言构造一个语法（函数式语言通过对表达式求值来完成计算工作）。**R**是一门用于统计学的编程语言，它在数据分析领域的应用日趋广泛。我挑选**R**语言作为示例是因为它的语法主

要由庞大的表达式规则组成。这是一个好机会，能加深我们对真实语言中运算符优先级和结合性的理解。

在牢牢掌握构造语法的知识之后，我们就可以在识别语言的基础上更进一步处理语言的内部逻辑：这些逻辑是应用程序遇到自身关心的输入文本时触发的动作。在下一章中，我们将编写语法分析器的监听器，它们能够建立相关数据结构、管理用于跟踪变量和函数定义的符号表，以及执行语言的翻译工作。

我们的学习将从CSV文件语法开始。

6.1 解析CSV文件

在5.3节中关于序列模式的内容中，我们已经见过了基本的CSV语法，现在让我们对它进行一些增强，使它能够识别标题行，并且允许空列存在。下面是一个典型的输入文件：

```
examples/data.csv
Details,Month,Amount
Mid Bonus,June,"$2,000"
,January,"""zippo"""
Total Bonuses,"","$5,000"
```

可以看到，标题行和常规行并无区别，我们只是将其中的列当作列的标题。为了从中提取出标题行，我们采用的方法是单独匹配它，而非使用`row+`匹配所有的行后再进行筛选。这样做的原因是，当需要基于

该语法构建一个真正的应用程序时，我们可能会希望对标题行进行区别对待。采用这种方法，我们就能对CSV文件的第一行进行特殊处理了。下面是语法的第一部分。

```
examples/CSV.g4
```

```
grammar CSV;
```

```
file : hdr row+ ;
```

```
hdr : row ;
```

为避免混淆，我们引入了一个名为**hdr**的新规则。虽然从语法角度看，标题行只是一个常规的行，但是，将它单独区分出来，使得它的角色更加清晰。你可以将它和**row+**或者**row row***做一下比较，体会其中的差异。

row规则和之前相同：一列由逗号分隔且由换行符终止的字段。

```
examples/CSV.g4
```

```
row : field (',' field)* '\r'? '\n' ;
```

为了让我们的字段定义比之前章节更加灵活，我们打算允许两个逗号之间出现任意的文本、字符串，甚至什么都没有。

```
examples/CSV.g4
```

```
field
```

```
    : TEXT
```

```
    | STRING
```

```
    |
```

```
    ;
```

TEXT类型的词法符号是下一个逗号或者换行符之前的任意字符序列。字符串是两个双引号之间的任意字符序列。下面是两个我们之前用过的词法符号定义：

```
examples/CSV.g4
```

```
TEXT : ~[, \n\r"]+ ;
```

```
STRING : '"' ( '"' | ~'"' )* '"' ; // 两个双引号是对双引号的转义
```

为了允许被双引号包围的字符串中出现双引号，CSV格式通常使用两个双引号来转义。这就是STRING规则的子规则（ `'"' | ~'"'` ）*存在的原因。我们不能使用通配符来构造非贪婪循环（ `'"' | ~'"'` ）*?，因为它一旦遇到在字符串开始之后的第一个`"`，便会终止匹配过程。类似`"x"y"`的输入就会被匹配成两个字符串，而非单个包含`"`的字符串。要记住，非贪婪的子规则是在保证整个父规则匹配成功的前提下，匹配数量尽可能少的字符。

在测试文法规则之前，让我们先看一下词法分析器生成的词法符号流，以保证它正确地对字符流进行了拆解。用grun别名来运行TestRig，带上选项-tokens，得到如下所示的输出：

```

⇒ $ antlr4 CSV.g4
⇒ $ javac CSV*.java
⇒ $ grun CSV file -tokens data.csv
< [@0,0:6='Details',<4>,1:0]
  [@1,7:7=',',<1>,1:7]
    [@2,8:12='Month',<4>,1:8]
      [@3,13:13=',',<1>,1:13]
        [@4,14:19='Amount',<4>,1:14]
          [@5,20:20='\n',<2>,1:20]
            [@6,21:29='Mid Bonus',<4>,2:0]
              [@7,30:30=',',<1>,2:9]
                [@8,31:34='June',<4>,2:10]
                  [@9,35:35=',',<1>,2:14]
                    [@10,36:43='$2,000',<5>,2:15]
                      [@11,44:44='\n',<2>,2:23]
                        [@12,45:45=',',<1>,3:0]
                          [@13,46:52='January',<4>,3:1]
                            ...

```

从这些词法符号看来，一切顺利。输出的标点符号、文本、字符串都和预期结果一致。

现在，看一下我们的语法是如何从输入的词法符号流中识别出语法结构的。使用**-tree**选项，测试组件就能够以文本形式打印出语法分析树（为阅读方便，进行了一些整理）。

```

⇒ $ grun CSV file -tree data.csv
< (file
  (hdr (row (field Details) , (field Month) , (field Amount) \n))
  (row (field Mid Bonus) , (field June) , (field "$2,000") \n)
  (row field , (field January) , (field ""zippo"" ) \n)
  (row (field Total Bonuses) , (field "" ) , (field "$5,000") \n)
)

```

其中根节点代表了起始规则**file**匹配到的语法结构。它有若干个代表数据行的子节点，且以标题行为首。这棵语法分析树的外观如图6-1所示（通过**-ps file.ps**选项获取）。

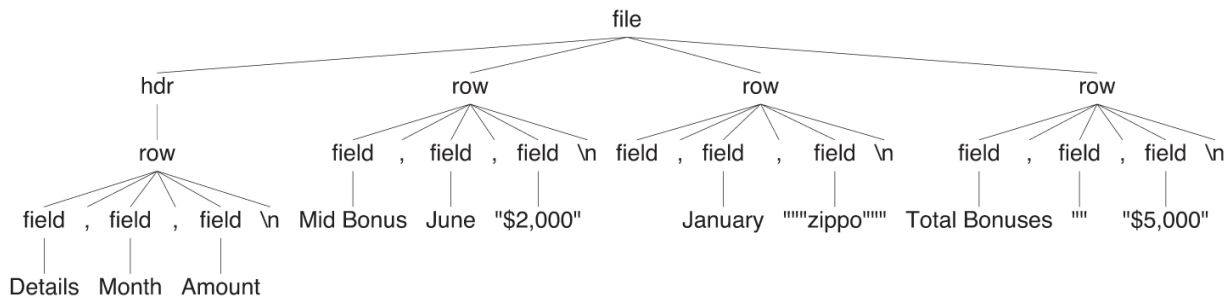


图6-1 语法分析树外观

因为简单，CSV是一种很好的数据存储格式。不过，如果我们需要在一个字段中存储多个值，它就无能为力了。对于这种情况，我们需要一种允许嵌套元素的数据格式。

6.2 解析JSON

JSON是一种存储键值对的数据格式，由于值本身也可以作为键值对的容器，JSON中可以包含嵌套结构。设计一个用于JSON的语法分析器让我们有机会基于一门语言的参考手册设计语法，并处理更加复杂的词法规则。为了使说明更加形象，下面是一个简单的JSON数据文件：

```

examples/t.json
{
    "antlr.org": {
        "owners" : [],
        "live" : true,
        "speed" : 1e100,
        "menus" : ["File", "Help\nMenu"]
    }
}

```

我们的目标是通过阅读JSON参考手册、查看它的语法描述图和现有的语法来构造一个能够解析JSON的ANTLR语法。我们将从JSON参考手册中提取关键词汇，然后一步步将它们编写成ANTLR规则。这个过程从语法结构开始。

1.JSON的语法规则

JSON语法指明，一个JSON文件可以是一个对象，或者是一个由若干个值组成的数组。从语法上看，这不过是一个选择模式，因此，我们可以用下列规则来表达：

```
examples/JSON.g4
json:    object
        |   array
        ;
```

下一步是将json规则引用的各个子规则进行分解。对于对象，JSON语法规是这样规定的：

一个对象是一组无序的键值对集合。一个对象以一个左花括号（{）开始，且以一个右花括号（}）结束。每个键后跟一个冒号（:），键值对之间由逗号分隔（,）。

JSON官方网站上的语法图强调对象中的键必须是字符串。

为将上面这一段自然语言的表述转换为语法结构，我们试着将它分解，从中提取关键的、能够指示采用何种模式的词组。第一句话中

的“一个对象是”明确地告诉我们创建一个名为`object`的规则。接着，“一组无序的键值对集合”实际上就是若干个“对”组成的序列。“无序的集合”指明了对象的键的语义，即键的顺序没有意义。这意味着，在语法分析的过程中，我们可以将它当作传统的键值对列表来匹配。

第二个句子引入了一个词法符号依赖，因为一个对象是以左右花括号作为开始和结束的。最后一个句子进一步指明了键值对序列的细节：由逗号分隔。至此，我们可以得到下列ANTLR标记编写的语法：

```
examples/JSON.g4
object
    :   '{' pair (',' pair)* '}'
    |   '{' '}' // 空对象
    ;
pair:   STRING ':' value ;
```

出于让语法清晰、减少重复代码的目的，最好将键值对拆分为单独的规则。否则，上述语法就变成了：

```
object : '{' STRING ':' value (',' STRING ':' value)* '}' | ... ;
```

注意，在其中我们将`STRING`当作一个词法符号，而非语法规则。这是因为，通常情况下，一个读取JSON的程序期望将字符串作为完整的实体处理，而非字符序列。这是我们在5.6节中提及的经验法则，字符串应该被当作词法符号处理。

JSON的语法参考中还包括一些非正式的语法规则，让我们来看看它和ANTLR规则有何差异。下列语法摘自JSON语法参考：

```
object
  {}
  { members }

members
  pair
  pair , members

pair
  string : value
```

和我们的规则一样，该语法规则也将`pair`规则单独拆了出来，不过，它包含一个我们没有的规则`members`。这是一种不使用 (...) *循环来表达序列模式的方式，详见下面的“循环vs.尾递归”。

对于JSON中的另外一种高级结构——数组，语法参考描述如下：

数组是一组值的有序集合。一个数组由一个左方括号开始 (`[`)，由一个右方括号 (`]`) 结束。其中的值由逗号 (`,`) 分隔。

和`object`规则一样，`array`包含一个由逗号分隔的序列模式和一个左右方括号间的词法符号依赖。

examples/JSON.g4

```
array
  : '[' value (',' value)* ']'
  | '[' ']' // 空数组
  ;
```

在上述规则的基础上进一步细化，我们就需要编写规则value，通过JSON语法参考中的描述我们可以知道，它是一个选择模式。

一个值可以是一个双引号包围的字符串、一个数字、true/false、null、一个对象，或者一个数组。这些结构中可能发生嵌套。

循环vs.尾递归

JSON参考手册中的members规则看上去非常奇怪，因为它很难用自然语言描述：在pair规则和pair后紧接着自身的规则中做出选择。

```
members
  pair
  pair , members
```

其中缘由在于，ANTLR支持扩展巴克斯-诺尔范式（EBNF）语法，而JSON参考手册中直接使用了巴克斯-诺尔范式（BNF）。BNF不支持类似（...）*的循环，因此，它们使用了尾递归（某个规则在最后一个备选分支的最后一个元素中调用了自身）来对循环进行模拟。

为了展示这种尾递归规则和自然语言表述之间的关系，下面是members规则派生出的一个、两个和三个pair的规则：

```
members => pair

members => pair , members
        => pair , pair

members => pair , members
        => pair , pair , members
        => pair , pair , pair
```

这再次论证了5.2节中提及的警告：将现有的语法当作指南，而非圣经。

其中，“嵌套”这个术语指明需要使用嵌套模式，因此，我们知道，**value**规则中会包含对递归规则的引用。使用ANTLR标记编写的**value**规则如下所示。

```
examples/JSON.g4
```

```
value
:   STRING
|   NUMBER
|   object // 递归调用
|   array  // 递归调用
|   'true' // 关键字
|   'false'
|   'null'
;
```

由于**value**规则引用了**object**和**array**，它成为（间接）递归规则。通过**value**调用二者中的任一，最终都会回到**value**规则。

value规则使用字符串常量来匹配JSON中的关键字。出于和字符串相同的原因，我们也把数字当作词法符号处理：程序通常认为数字是完整的实体。

这就是解析JSON的语法规则。至此，我们已经完全确定了JSON文件的结构，图6-2展示出我们的语法是如何解析之前的那份样例输入的。

当然，因为缺少词法分析器的相关规则，我们现在还无法通过程序来获取这幅图。我们还需要为两种关键的词法符号编写规则：**STRING**和**NUMBER**。

2.JSON词法规则

根据JSON语法参考，字符串定义如下：

一个字符串就是一个由零个或多个Unicode字符组成的序列，它由双引号包围，其中的字符使用反斜杠转义。单个字符由长度为1的字符串来表示。字符串和C/Java中的字符串非常相似。

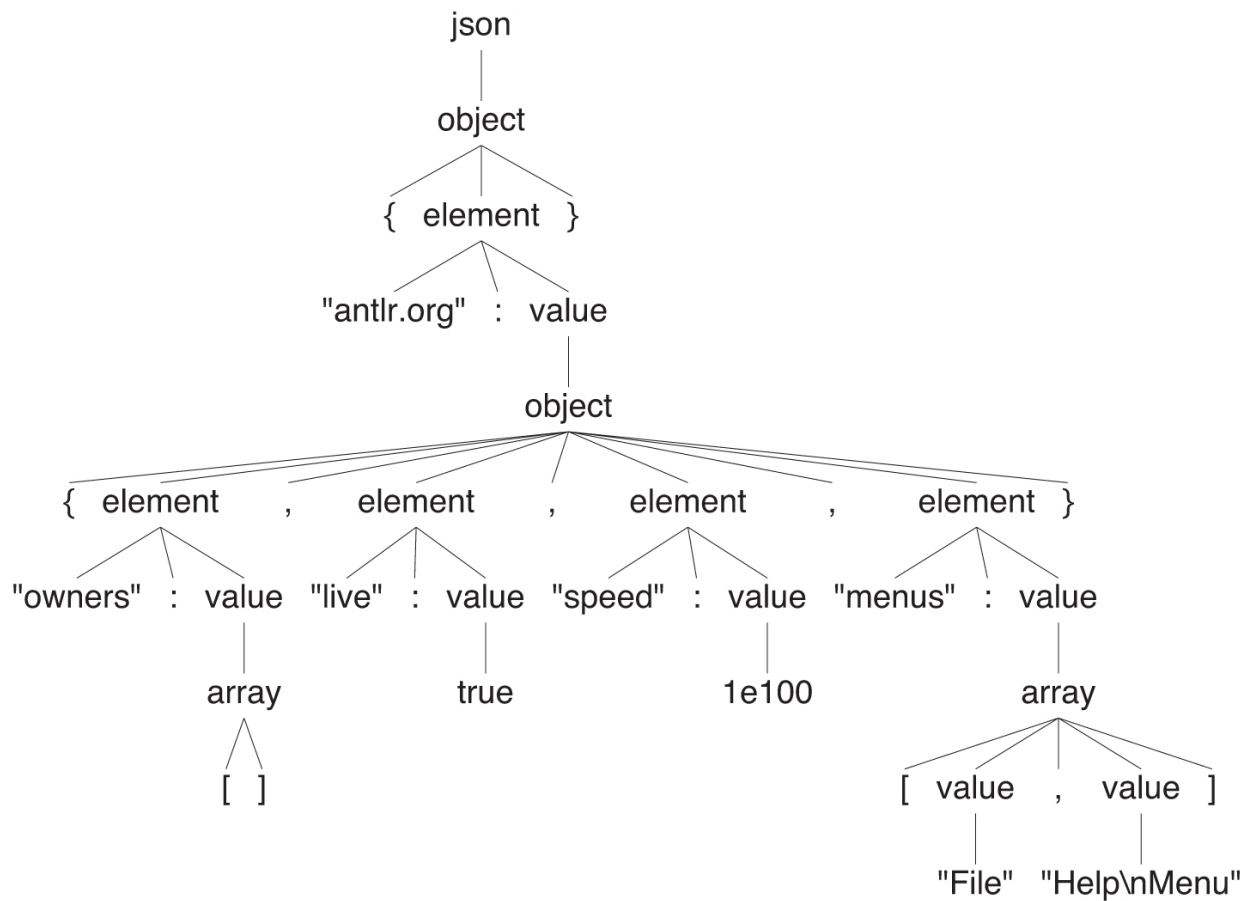


图6-2 语法解析示例

前已述及，大多数语言中的字符串都是非常相似的。JSON中的字符串和我们在5.5节的“匹配字符串常量”中讨论过的字符串非常相似，只不过JSON增加了对Unicode字符的转义。查看JSON语法参考，我们可以发现它对字符串的描述是不完备的。语法参考中的描述如下：

char

任意除 " 字符、\ 字符以及控制字符之外的 Unicode 字符

\"

\\

\/

\b

\f

\n

\r

\t

\u 由 4 位十六进制表示的数字

其中指明了需要被转义的字符，并且指明了我们应当匹配任意除双引号和反斜杠之外的字符。我们可以通过“反向选择序列”~["\\]来满足这个要求（~操作符的意思是“非”）。我们的STRING定义如下所示：

examples/JSON.g4

```
STRING : '"' (ESC | ~["\\])* ' ' ;
```

ESC规则匹配一个Unicode序列或者预定义的转义字符。

examples/JSON.g4

```
fragment ESC : '\\ ' (["\\ /bfnrt"] | UNICODE) ;
```

```
fragment UNICODE : 'u' HEX HEX HEX HEX ;
```

```
fragment HEX : [0-9a-fA-F] ;
```

在UNICODE规则中，我们定义了一个HEX片段规则作为简写，来代替需要多次重复的十六进制数字（以fragment开头的规则只能被其他的词法分析器规则使用，它们并不是词法符号）。

最后一个需要编写的词法符号是NUMBER。JSON语法参考对其定义如下：

一个数字和C/Java中的数字非常相似，除了一点之外：不允许使用八进制和十六进制格式。

JSON语法参考中对数字的描述稍显复杂，我们可以将它整理成三个主要的备选分支。

```
examples/JSON.g4
NUMBER
    :  '-'? INT '.' INT EXP?  // 1.35, 1.35E-9, 0.3, -4.5
    |  '-'? INT EXP           // 1e10 -3e4
    |  '-'? INT               // -3, 45
    ;
fragment INT :  '0' | [1-9] [0-9]* ; // 除 0 外的数字不允许以 0 开始
fragment EXP :  [Ee] [+\\-]? INT ; // \\- 是对 - 的转义，因为 [...] 中的 - 用于表达“范围”语义
```

同样，片段规则INT和EXP减少了重复代码，使语法可读性更强。

根据JSON语法参考，我们知道，INT不应当匹配除0之外的以0开头的数字。

```
int
    digit
    digit1-9 digits
    - digit
    - digit1-9 digits
```

我们在NUMBER规则中处理负号，这样，我们就能将精力集中于前两个备选分支：digit和digit1-9 digits。前者匹配任意的单个数字，所以0是可行的。后者匹配以1到9，即除0之外开头的数字。

和上一节中的CSV语法不同的是，JSON语法需要额外处理空白字符。

在任意两个词法符号之间，可以存在任意多的空白字符。

这就是空白字符的典型含义，所以我们可以复用上一章末尾的“入门专用词法规则”。

```
examples/JSON.g4
```

```
WS : [ \t\n\r]+ -> skip ;
```

现在，解析JSON的词法规则和语法规则都已就绪，我们可以将它们付诸实践了。首先，我们来打印输入文本[1, "\u0049", 1.3e9]中的词法符号。

```
⇒ $ antlr4 JSON.g4
⇒ $ javac JSON*.java
⇒ $ grun JSON json -tokens
⇒ [1, "\u0049", 1.3e9]
⇒ EoF
  ⚡ [@0,0:0=' ',<5>,1:0]
    [@1,1:1='1',<11>,1:1]
    [@2,2:2=',',<4>,1:2]
    [@3,3:10='"\u0049"',<10>,1:3]
    [@4,11:11=',',<4>,1:11]
    [@5,12:16='1.3e9',<11>,1:12]
    [@6,17:17=']',<1>,1:17]
    [@7,19:18='<EOF>',<-1>,2:0]
```

我们的词法分析器将输入流处理成了正确的词法符号流，因此我们可以进一步测试语法规则了。

```
⇒ $ grun JSON json -tree
⇒ [1, "\u0049", 1.3e9]
⇒ EoF
  ⚡ (json (array [ (value 1) , (value "\u0049") , (value 1.3e9) ]))
```

语法分析的结果显示，词法符号流被正确地解析成了三个值，看上去一切顺利。对于更复杂的语法，我们可能会使用许多输入文件来验证其正确性。

至此，我们已经为两种数据存储语言（CSV和JSON）编写了语法，下面让我们进一步讨论一门名为DOT的语言，它的语法更加复杂和棘手，并且引入了一种新的词法模式：不区分大小写的关键字。

6.3 解析DOT语言

DOT是一门声明式编程语言，主要用于描述网络图、树或者状态机之类的图形（DOT是声明式语言的原因是，我们描述的是图形及图形间的连接是什么，而非构造图形的过程）。它是一种应用广泛的图形工具，尤其是在你的程序需要生成图形时。例如，ANTLR的-atn选项就使用DOT来产生可视化的状态机。

为了能快速熟悉这门语言，假设我们需要将一个程序的四个函数间的调用关系可视化。我们可以手工画出来，或者使用下面的DOT代码来指定它们之间的关系（这些DOT代码可以手写或者编写一个源代码分析工具来自动生成）：

```
examples/t.dot
digraph G {
    rankdir=LR;
    main [shape=box];
    main -> f -> g;          // main 调用 f, f 调用 g
    f -> f [style=dotted] ; // f 是递归的
    f -> h;                  // f 调用了 h
}
```

图6-3就是使用DOT可视化工具graphviz生成的结果图。

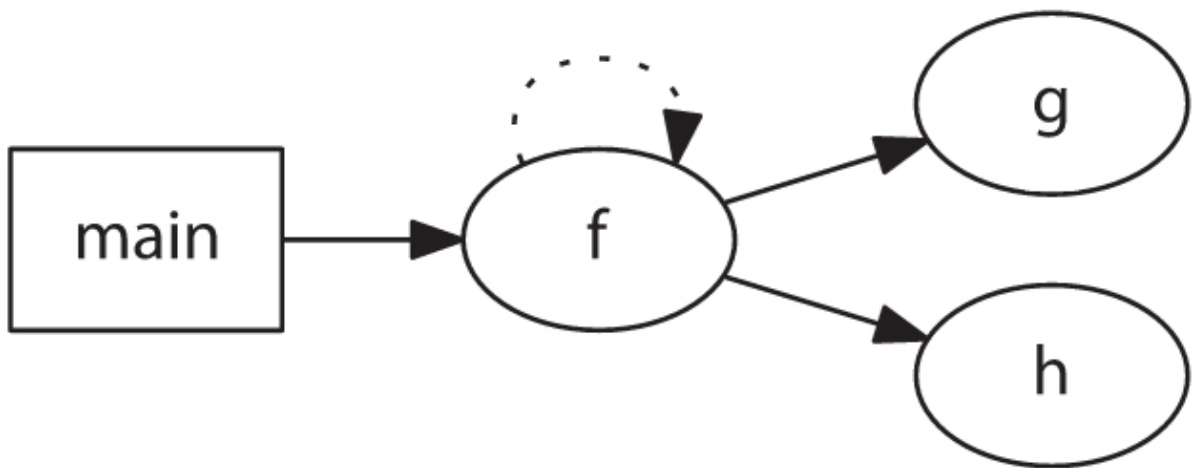


图6-3 使用DOT可视化工具生成的结果图

幸运的是，DOT语法指南中包含了几乎可以直接使用的句法规则，我们所需的仅仅是将它们翻译为ANTLR语法。不幸的是，我们需要自行编写词法规则。接下来，我们必须通过阅读文档和一些范例代码来完成全部规则的编写，为简单起见，我们从语法规则开始。

1.DOT语言的语法规则

下面是将DOT语言参考文档翻译为ANTLR标记的结果：

examples/DOT.g4

```
graph      :  STRICT? (GRAPH | DIGRAPH) id? '{' stmt_list '}' ;
stmt_list  :  ( stmt ';' ? ) * ;
stmt       :  node_stmt
              | edge_stmt
              | attr_stmt
              | id '=' id
              | subgraph
              ;
attr_stmt  :  (GRAPH | NODE | EDGE) attr_list ;
attr_list  :  ('[' a_list? ']') + ;
a_list     :  (id ('=' id)? ',') + ;
edge_stmt  :  (node_id | subgraph) edgeRHS attr_list? ;
edgeRHS    :  ( edgeop (node_id | subgraph) ) + ;
edgeop     :  '->' | '--' ;
node_stmt  :  node_id attr_list? ;
node_id    :  id port? ;
port       :  ':' id (':' id)? ;

subgraph   :  (SUBGRAPH id?)? '{' stmt_list '}' ;
id         :  ID
              | STRING
              | HTML_STRING
              | NUMBER
              ;
```

我们所做的唯一修改是port规则。参考文档中给出的规则如下：

```
port:      ':' ID [ ':' compass_pt ]
          |   ':' compass_pt
compass_pt
          :  (n | ne | e | se | s | sw | w | nw)
```

如果方位点（compass point）是一个关键字，即不能作为合法的标识符，那么上述规则就能够被正常使用。然而，参考文档中给出的说明改变了该语法的含义。

注意，合法的方位点的值并非关键字，因此这些字符串可被用于任何常规标识符能够出现的地方。

这意味着我们必须接受类似`n->sw`这样的极端情况，其中`n`和`sw`都不是关键字，而是标识符。参考文档接着指出：“...反之，语法分析器接受任意标识符”。这句话的含义不是非常清楚，似乎是语法分析器允许任意标识符作为方位点。如果这种猜测成立的话，我们在语法中就完全无须担心方位点的问题，我们可以用`id`替换掉参考文档中的`compass_pt`规则。

```
port:    ':' id (':' id)? ;
```

为确认这一点，最好使用一个DOT语言查看器来验证我们的假设，例如Graphviz网站上提供的工具。经验证，下列图定义能被正确无误地识别，因此我们的`port`规则是正确的。

```
digraph G { n -> sw; }
```

至此，我们拥有了全部的语法规则。假定我们已经完成了全部的词法符号定义，让我们看一看根据上文样例输入`t.dot`生成的语法分析树（使用`grun DOT graph-gui t.dot`命令），如图6-4所示。

现在，让我们来尝试定义这些词法符号。

2.DOT语言的词法规则

由于DOT语言参考指南没有给出正式的词法规则，我们就需要根据其中的描述来生成它们。不妨从最简单的关键字开始。

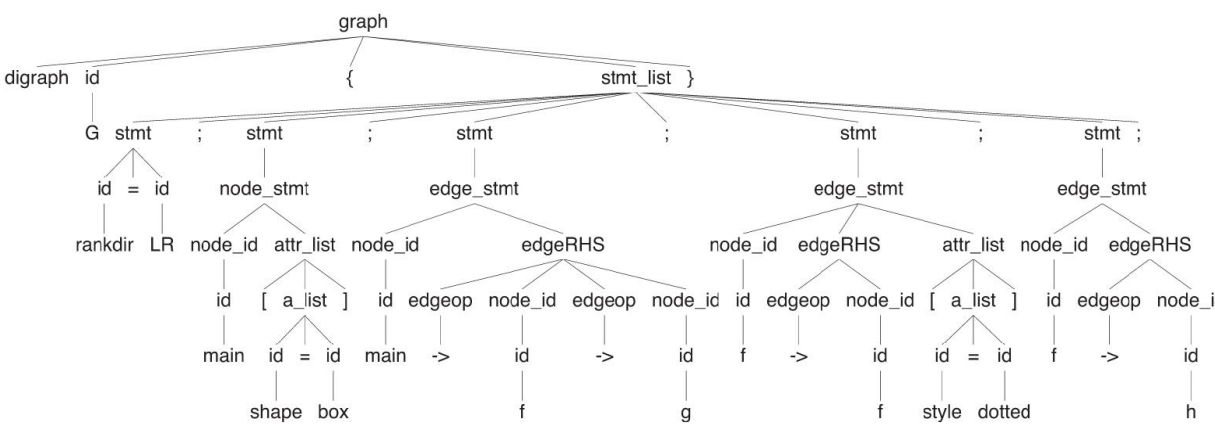


图6-4 输入t.dot生成的语法分析树

参考文档指出，“关键字node、edge、graph、digraph、subgraph，以及strict是不区分大小写的”。如果它们区分大小写，我们就可以在语法中简单地使用类似'node'的字符串常量。为了能够识别nOdE这样的变体，我们需要在关键字的词法规则中为每个字符分别指定大小写的形式。

examples/DOT.g4		
STRICT	:	[Ss][Tt][Rr][Ii][Cc][Tt] ;
GRAPH	:	[Gg][Rr][Aa][Pp][Hh] ;
DIGRAPH	:	[Dd][Ii][Gg][Rr][Aa][Pp][Hh] ;
NODE	:	[Nn][Oo][Dd][Ee] ;
EDGE	:	[Ee][Dd][Gg][Ee] ;
SUBGRAPH	:	[Ss][Uu][Bb][Gg][Rr][Aa][Pp][Hh] ;

DOT语言中的标识符和其他编程语言相似。

任意由字母表中的字符（`[a-zA-Z\200-\377]`）、下划线（`'_'`）或数字（`[0-9]`）组成的，不以数字开头的字符串。

八进制的数字范围`\200-\377`用十六进制来表示是`80`到`ff`，因此我们的ID规则如下：

```
examples/DOT.g4
ID          :   LETTER (LETTER|DIGIT)*;
fragment
LETTER      :   [a-zA-Z\u0080-\u00FF_] ;
```

我们定义了一个辅助规则**DIGIT**来匹配数字。参考文档指出，数字遵循下列正则表达式：

```
[ - ]? ( . [ 0 - 9 ] + | [ 0 - 9 ] + ( . [ 0 - 9 ] * ) ? )
```

使用**DIGIT**替换其中的`[0-9]`，我们就得到了用ANTLR标记编写的、代表DOT语言中的数字的规则，如下：

```
examples/DOT.g4
NUMBER      :   ' - ' ? ( ' . ' DIGIT + | DIGIT + ( ' . ' DIGIT * ) ? ) ;
fragment
DIGIT       :   [ 0 - 9 ] ;
```

DOT语言中的字符串定义较为基础。

任意的由双引号包围的字符序列（`"..."`），可能包括转义后的双引号`\"`。

我们使用点通配符来匹配字符串中的任意字符，直到遇见最后的双引号为止。额外地，我们将转义后的双引号作为子规则循环中的一个备选分支。

```
examples/DOT.g4
```

```
STRING      :   '\'' ( '\\\'|.|.)*? '\'' ;
```

DOT语言中还包含一种名为HTML字符串的元素，据我所知，它和字符串非常相似，唯一的差异在于它使用尖括号而不是双引号。参考文档中使用<...>来表示这种元素，描述如下：

在HTML字符串中，尖括号必须成对出现，其中可以包含未转义的换行符。除此之外，HTML字符串的内容必须是合法的XML，这就要求某些特殊字符（"、&、<以及>）需要被转义，以便嵌入XML标签的属性或者文本中。

上面的描述告诉了我们完成的工作，但是没有回答这一问题：我们是否可以在HTML注释中包含>。另外，它似乎暗示了我们需将标签序列放入尖括号中，类似<<i>hi</i>>。经DOT查看器试验，我们的猜测是正确的。从试验的结果看，DOT语言能够接受一对尖括号中的任意文本，只要这对尖括号是配对的。因此，HTML注释中的>不会被像XML解析器那样的处理方式忽略，即HTML字符串<foo<!--ksjdf>-->>会被当作字符串"foo<!--ksjdf>-->"处理。

我们可以使用ANTLR结构'<'.*? '>'来匹配两个尖括号之间的任意文本。不过，这个规则不允许其中出现嵌套的尖括号，因为它会把第一个>和第一个<配对，而不是我们期望的最近的<。下列规则能够达到预期效果：

```
examples/DOT.g4
/** " 在 HTML 字符串中，尖括号必须成对出现，其中可以
 * 包含未转义的换行符。"
 */
HTML_STRING :   '<' (TAG|~[<>])* '>' ;
fragment
TAG          :   '<' .*? '>' ;
```

其中的HTML_STRING规则允许TAG元素出现在配对的尖括号之间，这样就实现了一层的嵌套。~[<>]负责匹配类似“<”；的XML字符实体。它匹配除左右尖括号外的任何字符。在这里，我们不能使用通配符和非贪婪匹配循环，这是因为，若循环中的通配符能够匹配<foo，“(TAG|.)*?”就能匹配类似<<foo>的无效输入。即，如果使用了通配符，HTML_STRING无须调用匹配标签元素的TAG规则就能完成匹配，也就无法得到我们期望的结果。

你可能已经跃跃欲试，想到使用下面的递归来匹配尖括号了：

```
HTML_STRING : '<' (HTML_STRING|~[<>])* '>' ;
```

但是，这个规则匹配的是嵌套标签，而非将开始和结束的尖括号进行正确配对。一个嵌套的标签类似<<i
>>，这是我们所不希望看到

的。

DOT语言的最后一种词法结构是我们之前从未见过的。DOT语言匹配并丢弃以#开头的行，它认为那是C语言的预处理器的输出。我们可以用与之前类似的单行注释的方法来处理它们。

```
examples/DOT.g4
```

```
PREPROC      :   '#' .*? '\n' -> skip ;
```

这就是DOT语言的全部语法（虽然其中有些规则我们还不十分熟悉）。我们成功地编写了第一门复杂语言的语法！在本节中，除了更加复杂的词法结构和语法结构，还有一个重点需要强调：要想揭开一门语言的神秘面纱，我们需要分析不同来源的信息。语言的规模越大，我们需要的参考文档和各式各样的范例代码就越多。有时候，只有设法对语言现有的实现进行试探，才能发现边界情况。语言的参考文档通常并非一目了然。

此外，我们必须决定语法分析过程分解为哪些步骤，以及在每个步骤中，哪些部分可以暂时搁置，留待后续处理。提及的这些要点的例子如，我们将特殊的方位点ne和sw当作普通标识符，以此来测试解析器的结果。又如，我们编写语法时，并不理解<...>中的HTML字符串的含义。一个DOT语言的完整实现最终需要验证并处理这些元素，但是语法分析器可以仅仅将他们当作数据块来处理。

现在，是时候处理一些编程语言了。下一节中，我们将会为传统的、类似C语言的命令式编程语言编写语法。之后，我们将接受一个迄今为止最大的挑战：处理函数式编程语言R。

6.4 解析Cymbol语言

接下来，我们将为一门我自己设计的、名为Cymbol的语言编写一个语法，以此来展示类C语言的解析过程。Cymbol是一门简单的、非面向对象的编程语言，外观类似不带结构体的C语言。如果你想要自己创造一门新的编程语言的话，Cymbol的语法可以作为它的原型。本节不会介绍新的ANTLR语法结构，不过，我们编写的Cymbol语法将会展示如何构造一条左递归的表达式规则。

当设计一门新的语言时，我们就没有可以参考的正式语法和文档了。取而代之的是，我们通过设计新语言的范例代码来起步。之后，我们就可以按照5.1节中的方法，从范例代码中提取语法（这也是我们处理缺乏正式语法和参考文档的现有语言的方式）。下面一段带有全局变量和递归函数声明的程序就是Cymbol代码：

```
examples/t.cymbol
// Cymbol test
int g = 9;          // 全局变量
int fact(int x) { // 求阶乘的函数
    if x==0 then return 1;
    return x * fact(x-1);
}
```

就像厨艺展示一样，让我们先看看最终的成品，以便将目标铭记于心。图6-5的语法分析树展示了最终的语法应当如何解析输入的代码（通过`grun Cymbol file-gui t.cymbol`）。

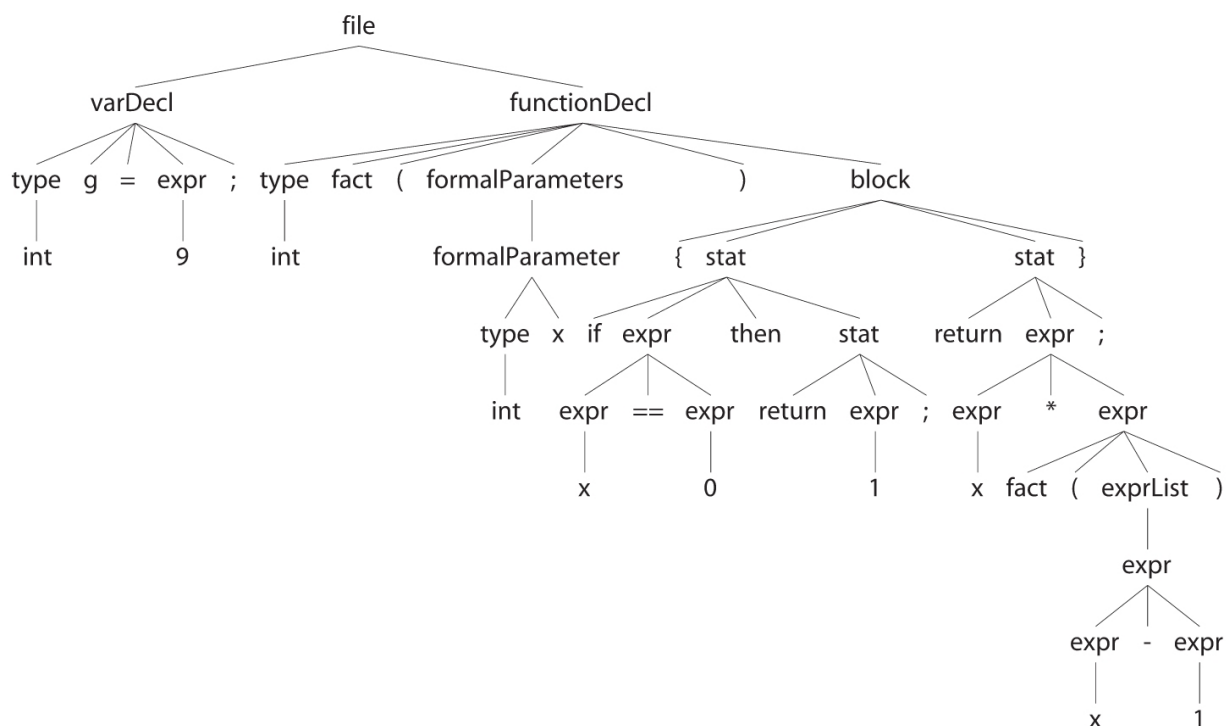


图6-5 展示了最终语法应当如何解析输入代码的语法分析树

从最粗的粒度观察Cymbol程序，我们可以发现它由一系列全局变量和函数声明组成。

`examples/Cymbol.g4`

```
file: (functionDecl | varDecl)+ ;
```

同所有的类C语言一样，变量声明由一个类型开始，随后是一个标识符，最后是一个可选的初始化语句。

examples/Cymbol.g4

```
varDecl
    :   type ID ('=' expr)? ';'
    ;
type:   'float' | 'int' | 'void' ; // 用户定义的类型
```

函数声明也基本上相同：类型后面跟着函数名，随后是被括号包围的参数列表，最后是函数体。

examples/Cymbol.g4

```
functionDecl
    :   type ID '(' formalParameters? ')' block // "void f(int x) {...}"
    ;
formalParameters
    :   formalParameter (',' formalParameter)*
    ;
formalParameter
    :   type ID
    ;
```

一个函数体是由花括号包围的一组语句。让我们先构造六种语句：嵌套的代码块、变量声明、if语句、return语句、赋值语句，以及函数调用。我们可以用下面的ANTLR语法来表达它们：

examples/Cymbol.g4

```
block: '{' stat* '}' ; // 语句组成的代码块，可以为空
stat: block
    | varDecl
    | 'if' expr 'then' stat ('else' stat)?
    | 'return' expr? ';'
    | expr '=' expr ';' // 赋值
    | expr ';'          // 函数调用
    ;
```

Cymbol语言的最后一个主要部分是表达式语法。因为Cymbol实际上仅仅是其他语言的原型或者基础，因此没有必要包含非常多的运算符。

假设我们的表达式包括一元取反、布尔非、乘法、加法、减法、函数调用、数组索引、等同性判断、变量、整数以及括号表达式。

```
examples/Cymbol.g4
expr:  ID '(' exprList? ')' // 类似 f(), f(x), f(1,2) 的函数调用表达式
      |  expr '[' expr ']' // 类似 a[i], a[i][j] 的数组索引表达式
      |  '-' expr          // 一元取反表达式
      |  '!' expr          // 布尔非表达式
      |  expr '*' expr
      |  expr ('+'|'-') expr
      |  expr '==' expr    // 等同性判断表达式（它是优先级最低的运算符）
      |  ID                // variable reference
      |  INT
      |  '(' expr ')'
      ;
exprList : expr (',' expr)* ; // 参数列表
```

其中的重点是我们通常将备选分支按照从高到低的优先级进行排序

（有关ANTLR移除左递归和处理运算符优先级的深入讨论见第14章）。为了说明运算符优先级的应用，我们来查看一下“-x+y;”和“-a[i];”对应的语法分析树，它们的起始规则都是stat规则（不使用file规则是为了避免凌乱），如图6-6所示。

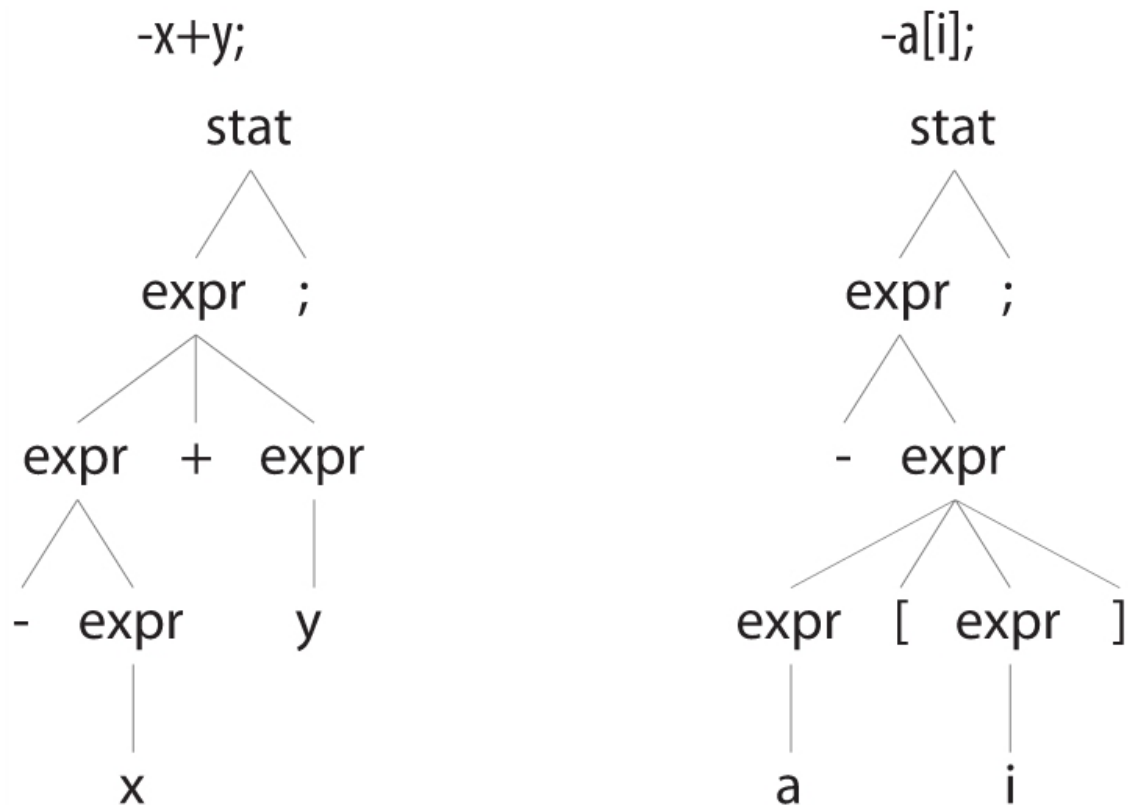


图6-6 起始规则均为stat规则的语法分析树

左侧的语法分析树显示了一元取反运算符和`x`紧密地结合在一起，因为它比加法的优先级更高。这是由于取反表达式的备选分支在加法表达式之前。另一方面，由于取反表达式的备选分支在数组索引表达式之后，取反运算符的优先级比数组索引运算符低。右侧的语法分析树显示，取反运算符被应用在`a[i]`之上，而非标识符`a`。在下一节中，我们将看到更加复杂的表达式规则。

我们不再循规蹈矩地编写词法规则，因为它们不能教会我们任何新知识。这些规则几乎是从前一章中词法模式一节中照搬过来的。在本节

中，我们主要将注意力集中在探索命令式编程语言的语法结构上。

在本节中，我们基本上是在凭直觉构造一门“不带类的Java语言”，这使得编写Cymbol语言语法的过程异常顺利。如果你能够完全理解它，那么对你而言，创造一门属于你自己的复杂的命令式语言就是一件轻而易举的事情。

在下一节中，我们要走向另外一个极端——根据多种多样的参考文档、范例程序以及现有的R语言的实现，提炼出精确的语言结构，从而完成一个优秀的R语言语法。

6.5 解析R语言

R是一门极富表现力的领域特定（domain-specific）编程语言，专门用于描述和解决统计学问题。例如，在R语言中，新建向量、对向量调用函数、筛选向量都十分容易（下面的示例使用了R语言命令行交互工具）。

```
⇒ x <- seq(1,10,.5)      # x = 1, 1.5, 2, 2.5, 3, 3.5, ..., 10
⇒ y <- 1:5               # y = 1, 2, 3, 4, 5
⇒ z <- c(9,6,2,10,-4)    # z = 9, 6, 2, 10, -4
⇒ y + z                  # 将两个向量相加
  < [1] 10  8  5 14  1    # 结果是一个一维向量
⇒ z[z<5]                 # 所有满足 z < 5 的元素
  < [1]  2 -4
⇒ mean(z)                 # 计算向量 z 的均值
  < [1] 4.6
⇒ zero <- function() { return(0) }
⇒ zero()
  < [1] 0
```


R语言是一门中等大小却十分复杂的语言，并且，在我们之中的大多数人面前都有一道鸿沟：我们不了解R语言。这就意味着，我们无法像上一节的Cymbol那样，凭着对语言结构的内在感觉来完成语法的编写。我们必须根据参考手册、范例代码以及现有实现中正式的yacc语法，提炼出R语言的结构。

首先，最好通过一些综述来大致了解一下R语言。同时，我们也应当看一些R语言的代码来找找感觉，然后将它们作为最终的“验收测试”。

Ajay Shah已经编写了许多可用的范例代码。覆盖这些范例意味着我们的语法能够处理大部分R语言代码了（想要在不了解一门语言的情况下编写出针对该语言的完美语法是不可能的）。在R语言的官方网站上，有很多文档可以帮助我们完成编写R语言语法的工作，我们将主要使用其中的R-info和语言定义R-lang。

和往常一样，我们的语法编写从最粗的粒度开始。显然，站在整体的角度上看，R语言的程序由一系列表达式或者赋值语句构成。每个函数定义都是赋值语句，它等价于将一个函数赋值给一个变量。唯一令我们感到陌生的是，在R语言中存在三种赋值运算符：`<-`、`=`和`<<-`。就我们的目标而言，我们无需关心这些运算符的含义，因为我们要构建的仅仅是语法分析器，而非解释器或者编译器。我们对R语言程序结构的第一次分解如下所示：

```

prog : (expr_or_assign '\n')* EOF ;

expr_or_assign
:    expr ('<-' | '=' | '<<-' ) expr_or_assign
|    expr
;

```

通过阅读范例代码我们发现，R语言似乎允许在一行中存在多个分号分隔的表达式。R-intro文档确认了这一点。此外，虽然没有在参考手册中提及，R语言的命令行允许且会自动忽略输入的空行。将这些已知规则组合在一起，我们就得到了下列语法：

```

examples/R.g4
prog:  (    expr_or_assign (';'|NL)
        |    NL
        )*
      EOF
      ;

expr_or_assign
:    expr ('<-' | '=' | '<<-' ) expr_or_assign
|    expr
;

```

我们使用词法符号NL而不是常量'\n'的原因是我们希望同时允许Windows风格的换行符（\r\n）和UNIX风格的换行符（\n），而这很容易在词法规则中定义。

```

examples/R.g4
// Match both UNIX and Windows newlines
NL      :   '\r'? '\n' ;

```

注意，NL规则并没有像往常一样丢弃对应的词法符号。语法分析器将这些词法符号当作表达式的终止符，类似Java中的分号，因此，词法分析器必须将它们完整地输送给语法分析器。

R语言语法中的大部分内容是和表达式相关的，因此在本节的剩余部分我们将集中精力处理它们。在R语言中，有三种主要的表达式：语句表达式（**statement expression**）、运算符表达式（**operator expression**）和函数相关表达式（**function-related expression**）。由于R语言的语句和其他命令式编程语言的对应部分非常相似，我们首先完成这部分工作。下面是**expr**规则中包含的备选分支（它们位于运算符表达式的备选分支之后）：

```
examples/R.g4
| '{' exprlist '}' // 复合语句
| 'if' '(' expr ')' expr
| 'if' '(' expr ')' expr 'else' expr
| 'for' '(' ID 'in' expr ')' expr
| 'while' '(' expr ')' expr
| 'repeat' expr
| '?' expr // 获取 expr 的帮助信息，通常是字符串或者标识符
| 'next'
| 'break'
```

其中，第一个备选分支匹配R-intro中提到的“表达式组”——“多条命令可以通过花括号（{和}）组成一个复合表达式”。下面是**exprList**规则的定义：

```
examples/R.g4
```

```
exprlist
:   expr_or_assign ((';'|NL) expr_or_assign?)*
|
;
```

R语言中的大多数表达式包含丰富的运算符。为了得到这些表达式的正确形式，最好的方法是参照yacc的语法。可执行的代码通常是（但并非总是）表达语言作者意图的最好向导。为了获知各运算符的优先级，我们需要查看优先级表，它显式地指明了相关运算符的优先级。例如，下列yacc语法给出了算术运算符定义（先列出的%left命令优先级较低）：

```
%left      '+' '-'
%left      '*' '/'
```

R-lang文档中有一个名为“中缀和前缀运算符”的章节，给出了运算符优先级规则，不过，它似乎漏掉了yacc语法中的：：：运算符。将所有一切组合在一起，我们就得到了下列处理二元、前缀和后缀运算符的表达式规则：

```
examples/R.g4
```

```
expr:   expr '[' sublist ']' ']' // '[' 源于 R 语言的 yacc 语法
      |   expr '[' sublist ']'
```

```

|  expr ( '::' | ':::') expr
|  expr ('$' | '@') expr
|  expr '^<assoc=right> expr
|  ('-' | '+') expr
|  expr ':' expr
|  expr USER_OP expr // 任意被 % 包围的文本 : '%' .* '%'
|  expr ('*' | '/') expr
|  expr ('+' | '-') expr
|  expr ('>' | '>=' | '<' | '<=' | '==' | '!=') expr
|  '!' expr
|  expr ('&' | '&&') expr
|  expr ('|' | '||') expr
|  '~' expr
|  expr '~' expr
|  expr ('->' | '->>' | ':=' ) expr

```

我们无须关心运算符的具体含义，因为在这个例子中，我们暂时只关心识别问题。我们唯一需要确保的是，我们的语法能够使用正确的优先级和结合性完成匹配。

`expr`规则的一个不同寻常的地方是，在备选分支`['sublist']`中，我们使用的是`['`而非`['`。（`[[...]]`得到的是包含单一元素的列表，而`[...]`生成一个子列表。）其中的`['`规则直接取自R语言的yacc语法。这可能是因为语法的作者希望强制“两个左方括号之间没有任何空白字符”，不过这一点并没有在参考手册中体现出来。

其中，`^`运算符后面带有后缀`<assoc=right>`（详见5.4节），因为R-lang文档指出：

指数运算符'^'以及向左赋值运算符'<-=<<-'自右向左组合，其他的运算符都是自左向右分组合的。即， 2^{2^3} 的结果是 2^8 ，而不是 4^3 。

在语句表达式和运算符表达式的规则都已经就绪之后，让我们来看expr规则的最后一部分：定义和调用函数。我们可以写出如下所示的备选分支：

```
examples/R.g4
| 'function' '(' formlist? ')' expr // 定义函数
| expr '(' sublist ')' // 调用函数
```

formlist和sublist规则分别定义了形式参数列表和实际参数列表。这两个规则的名字取自yacc语法，这样就能更加容易地对比两份语法。

formlist规则表达的函数形参遵循R-lang文档中的如下章节：

...一个由逗号分隔的元素组成的列表，每个元素都可以是标识符或者'identifier=default'的形式，或者是特殊的词法符号'...'。其中，default可以是任意的合法表达式。

我们可以编写一条和yacc语法中的formlist规则相似的ANTLR规则来实现它。

```
examples/R.g4
formlist : form (',' form)* ;
```

```

form:  ID
      |  ID '=' expr
      |  '...'
      ;

```

对于函数的调用，R-lang描述了参数表的语法，如下所示。

每个参数都可以带上标记（`tag=expr`），或者只是一个简单的表达式。参数可以为空或特殊词法符号之一，如`'...'`，`'..2'`等。

yacc语法额外增加了几条规则，它指出，参数可以是类似`"n"=0`，`n=1`，以及`NULL=2`的东西。这样，我们就得到了下列指定函数的调用参数的规则：

```

examples/R.g4
sublist : sub (',' sub)* ;
sub :
    |  expr
    |  ID '='
    |  ID '=' expr
    |  STRING '='
    |  STRING '=' expr
    |  'NULL' '='
    |  'NULL' '=' expr
    |  '...'
    ;

```

你可能有点疑惑，为什么我们不在`sub`规则中匹配`..2`这样的特殊词法符号？这是因为，我们无须显式地匹配它们，词法分析器会将它们当作标识符处理。根据R-lang文档：

标识符包含字母、数字、句点 ('.')，以及下划线。合法的标识符不能以数字、下划线和句点后的数字开头。...注意，以句点开头的标识符，例如'...', '..1', '..2'等，是特殊标识符。

为了满足上述要求，我们使用如下的标识符规则：

```
examples/R.g4
ID : '.' (LETTER|'_'|'.') (LETTER|DIGIT|'_'|'.')*
    | LETTER (LETTER|DIGIT|'_'|'.')*
    ;
fragment LETTER : [a-zA-Z] ;
```

其中，第一个备选分支将以句点开头的标识符的情况区分了出来。我们必须确保数字不是这种标识符的第二个字符，这是通过子规则

(LETTER|'_'|'.') 来完成的。此外，为了保证标识符不以数字或者下划线开头，我们的第二个备选分支以LETTER开头。对于..2这种情况，第一个备选分支会匹配它。它开头的'.'匹配该规则中的第一个点，子规则(LETTER|'_'|'.')匹配了它的第二个点，最后的那部分子规则匹配了数字2。

词法规则的其他部分是直接从之前章节中拷贝或者稍加扩展而来的，这里不再赘述。

让我们使用grun命令分析下面的输入文件，来一睹我们大作的风采。

```
examples/t.R
addMe <- function(x,y) { return(x+y) }
addMe(x=1,2)
r <- 1:5
```


图6-7是根据输入文件t.R建立并显示可视化的语法分析树的步骤。

```
$ antlr4 R.g4
$ javac R*.java
$ grun R prog -gui t.R
```

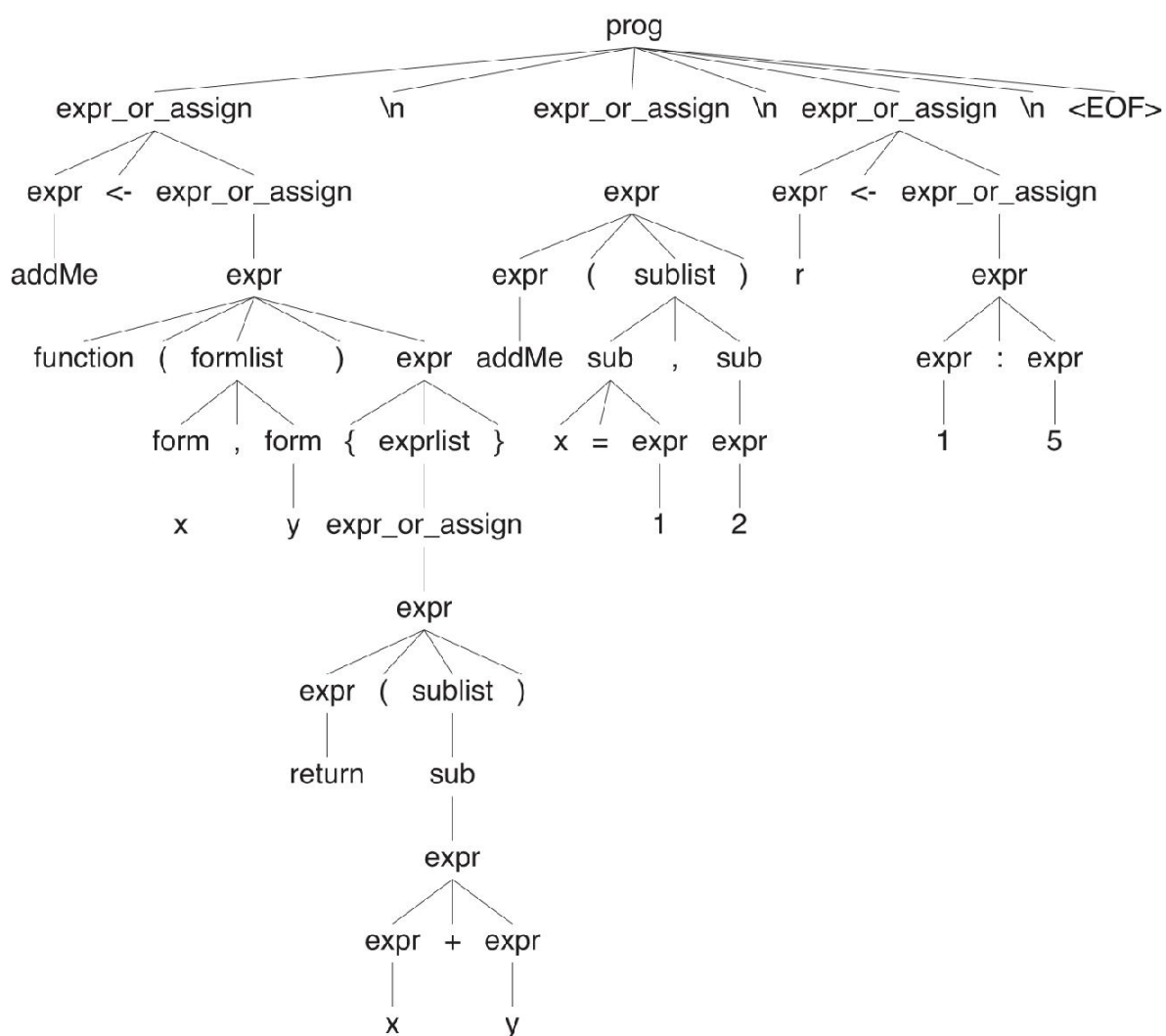


图6-7 输入文件t.R的语法分析树

只要每个表达式都像函数`addMe()`一样占一行，我们的R语法就能正常工作。不过，这个限制是不应该存在的，因为R语言允许函数和其他表达式占据多行。尽管如此，我们的任务（覆盖R语言本身的语法结构）已经圆满完成了。在源文件夹`code/extras`中，你可以看到解决该问题的方案，请参阅`R.g4`、`RFilter.g4`以及`TestR.java`。该方案是这样的：在词法分析器产生词法符号流后，根据R语言语法的要求，对其进行过滤，用适当的方法保留或者丢弃一些换行符。

在本章中，我们的目标是巩固ANTLR语法的相关知识，以及学习如何根据一门语言的参考文档、范例代码和非ANTLR语法来编写该语言的语法。到目前为止，我们已经研究了两门数据描述语言（CSV和JSON）、一门声明式语言（DOT）、一门命令式语言（Cymbol）和一门函数式语言（R）。这些示例覆盖了编写复杂语言语法所需知识的方方面面。不过，在继续学习之前，你最好通过下载语法并稍事改动来巩固这些知识。例如，你可以试着为Cymbol语言语法增加更多的运算符和语句。使用TestRig来验证你修改后的语法和范例代码之间的关系。

本书迄今为止的篇幅都是有关语言识别的。但是，语法本身只能反映出输入文本是否遵循该语言的规则。现在，我们都已经成为语法分析器领域的专家，在下一章中，我们将学习如何把应用程序的逻辑代码

和语法分析机制整合。之后，我们就能构建真正的语言类应用程序了。

第7章 将语法和程序的逻辑代码解耦

在之前的学习中，我们已经知道了如何使用ANTLR来定义语言的正式语法，现在，是时候对语法进行一些深入研究了。通常单独的语法并没有用处，而与其相关的语法分析器才能告诉我们输入语句是否遵循该语言的规范。为了构建一个语言类应用程序，语法分析器需要在遇到特定的输入语句、词组或者词法符号时触发特定的行为。这样的词组→行为的集合构成了我们的语言类应用程序，或者，至少担任了语法和外围程序间接口的角色。

在本章中，我们将会学习如何使用语法分析树监听器和访问器来构建语言类应用程序。监听器能够对特定规则的进入和退出事件（即识别到某些词组的事件）作出响应，这些事件分别由语法分析树遍历器在开始和完成对节点的访问时触发。另外，ANTLR自动生成的语法分析树也支持广为人知的访问者模式，从而允许程序控制语法分析树的遍历过程。

监听器和访问器机制的最大区别在于，监听器方法不负责显式调用子节点的访问方法，而访问器必须显式触发对子节点的访问以便树的遍历过程能够正常进行（正如我们在2.5节中看到的那样）。因为访问器

机制需要显式调用方法来访问子节点，所以它能够控制遍历过程中的访问顺序，以及节点被访问的次数。为简便起见，在下文中我会用术语“事件方法”（**event method**）来代替监听器的回调方法和访问器方法。

本章中，我们的目标是准确理解**ANTLR**自动生成的语法分析树遍历机制的工作方式和原理。我们将首先了解监听器机制的起源，以及如何使用监听器和访问器机制来使得程序逻辑代码与语法分离。随后，我们将会学习如何令**ANTLR**产生更加精确的事件——为规则的每个备选分支都生成一个事件。在深入了解**ANTLR**的语法分析树遍历机制后，我们将阅读三个计算器的实现代码，它们展示了传递子表达式结果的不同方法。最后，我们将会讨论三种方法的优缺点。到那时，我们就能胸有成竹地处理下一章中的真实语言了。

7.1 从内嵌动作到监听器的演进

如果你曾经使用过**ANTLR**的早期版本或者其他能够自动生成语法分析器的工具，你会惊讶于这一事实：我们构建语言类应用程序时可以在语法中内嵌动作（代码）。监听器和访问器机制能够将语法和程序逻辑代码解耦，从而大有裨益。这样的解耦将程序封装起来，避免了杂乱无章地分散在语法中。如果语法中没有内嵌动作，我们就可以在多个程序中复用同一个语法，而无须为每个目标语法分析器重新编译一次。

受益于内嵌动作的机制，ANTLR能基于同一个语法文件，使用不同的编程语言生成语法分析器（在ANTLR 4.0发布后，我参与了对不同目标语言提供支持的相关工作）。同时，在集成过程中，由于无须担心合并后内嵌动作的冲突，对语法的更新和bug修复也十分容易。

本节主要研究从包含内嵌动作的语法到完全与动作解耦的语法的演进过程。下列语法用于读取属性文件，这些文件的每行都是一个赋值语句，其中<<...>>是内嵌动作的概要。类似<<start file>>的标记代表一段恰当的Java代码。

```
grammar PropertyFile;
file : {<<start file>>} prop+ {<<finish file>>} ;
prop : ID '=' STRING '\n' {<<process property>>} ;
ID    : [a-z]+ ;
STRING : '"' .*? '"' ;
```

这样的紧耦合使得语法被绑定到了特定的程序上。更好的方案是，从ANTLR自动生成的语法分析器PropertyFileParser派生出一个子类，然后将内嵌动作转换为方法。这样的重构可以使得语法中仅仅包含方法调用，之后我们就可以通过语法分析器的子类实现任意数量的不同功能的程序，而无须修改原先的语法。下列代码展示了这样的重构过程：

```

grammar PropertyFile;
@members {
    void startFile() { } // 空实现
    void finishFile() { }
    void defineProperty(Token name, Token value) { }
}
file : {startFile();} prop+ {finishFile();} ;
prop : ID '=' STRING '\n' {defineProperty($ID, $STRING)} ;
ID    : [a-z]+ ;
STRING : '"' .*? '"' ;

```

上述解耦方案允许该语法被不同程序复用，但是由于方法调用的存在，它仍然和Java绑定在一起。我们随后会解决这个问题。

为了展示重构后的语法拥有良好的复用性，让我们构建两个不同的“语言类应用程序”，先从其中一个开始：在遇到属性时将它们打印出来。编写这个过程非常简单，只需继承ANTLR自动生成的语法分析器类，然后覆盖语法中触发的一个或多个方法即可。

```

class PropertyFilePrinter extends PropertyFileParser {
    void defineProperty(Token name, Token value) {
        System.out.println(name.getText()+"="+value.getText());
    }
}

```

需要注意的是，我们无须覆盖startFile () 和finishFile () 方法，因为ANTLR自动生成的PropertyFileParser已经提供了它们的默认实现。接下来，只需新建一个我们自定义的PropertyFilePrinter子类的实例，即可运行这个程序。

```
PropertyFileLexer lexer = new PropertyFileLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
PropertyFilePrinter parser = new PropertyFilePrinter(tokens);
parser.file(); // 运行我们的特殊版本的语法分析器
```

至于第二个程序，我们要完成的功能是将属性放入一个**Map**，而非打印出来。这个过程中，我们要做的仅仅是实现一个新的子类，然后为 `defineProperty ()` 加入不同的功能。

```
class PropertyFileLoader extends PropertyFileParser {
    Map<String,String> props = new OrderedHashMap<String, String>();
    void defineProperty(Token name, Token value) {
        props.put(name.getText(), value.getText());
    }
}
```

在这个语法分析器执行后，`props`字段将会包含属性文件的全部键值对。

这份语法仍然存在缺陷：受内嵌动作的限制，它只能生成Java编写的语法分析器。为了使语法可被重用并具有语言中立性，我们需要完全避免内嵌动作的存在。接下来的两节详细讲述了如何使用监听器和访问器来达到这个目的。

7.2 使用语法分析树监听器编写程序

构建应用逻辑和语法松耦合的语言类应用程序的关键在于，令语法分析器建立一棵语法分析树，然后在遍历该树的过程中触发应用逻辑代码。我们可以使用自己熟悉的方法遍历这样的语法分析树，也可以利

用ANTLR自动生成的树遍历器。在本节中，我们将会使用ANTLR内置的ParseTreeWalker构建一个与上一节类似的、基于监听器的属性文件处理程序。

让我们从一个“干净”的、识别属性文件的语法开始。

```
listeners/PropertyFile.g4
file : prop+ ;
prop : ID '=' STRING '\n' ;
```

下面是一个属性文件的样例：

```
listeners/t.properties
user="parrt"
machine="maniac"
```

基于上述语法，ANTLR生成了PropertyFileParser，它能够自动建立如图7-1中所示的语法分析树。

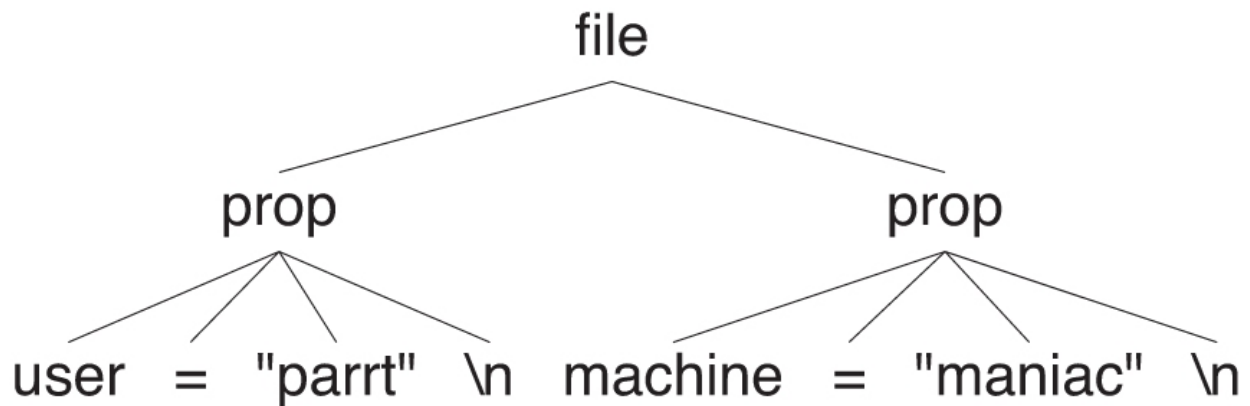


图7-1 自动建立的语法分析树

在得到了语法分析树之后，我们就能使用**ParseTreeWalker**来访问它的全部节点，触发这些节点上的**enter**和**exit**方法。

让我们看看**ANTLR**基于语法**PropertyFile**生成的监听器接口

PropertyFileListener。ANTLR的**ParseTreeWalker**在每次访问和离开节点的时候会分别触发对应规则子树的**enter**和**exit**方法。因为语法**PropertyFile**中只有两条语法规则，所以**PropertyFileListener**接口中有四个方法。

```
listeners/PropertyFileListener.java
```

```
import org.antlr.v4.runtime.tree.*;
import org.antlr.v4.runtime.Token;

public interface PropertyFileListener extends ParseTreeListener {
    void enterFile(PropertyFileParser.FileContext ctx);
    void exitFile(PropertyFileParser.FileContext ctx);
    void enterProp(PropertyFileParser.PropContext ctx);
    void exitProp(PropertyFileParser.PropContext ctx);
}
```

FileContext和**PropContext**类是每条语法规则对应的语法分析树节点的实现。它们包含一些很有用的方法，我们稍后将会深入探究。

为方便起见，ANTLR自动生成了一个名为**PropertyFileBaseListener**的默认实现，它包含了所有方法的空实现，即我们在上一节涉及语法的**@members**区域中手工编写的代码。

```

public class PropertyFileBaseVisitor<T> extends AbstractParseTreeVisitor<T>
    implements PropertyFileVisitor<T>
{
    @Override public T visitFile(PropertyFileParser.FileContext ctx) { }
    @Override public T visitProp(PropertyFileParser.PropContext ctx) { }
}

```

这样的默认实现允许我们只覆盖那些我们所关心的方法。例如，下面的属性文件加载器和之前一样包含单个方法，但是使用了监听器机制：

listeners/TestPropertyFile.java

```

public static class PropertyFileLoader extends PropertyFileBaseListener {

    Map<String,String> props = new OrderedHashMap<String, String>();
    public void exitProp(PropertyFileParser.PropContext ctx) {
        String id = ctx.ID().getText(); // prop : ID '=' STRING '\n' ;
        String value = ctx.STRING().getText();
        props.put(id, value);
    }
}

```

该版本和之前版本的最大差别在于，它继承了监听器基类（**base listener**）而非继承语法分析器，另外，监听器方法是在语法分析器完成解析之后才被触发的。

上文中出现了很多接口和类，所以让我们来看一下它们之间的继承关系（接口用斜体表示），如图7-2所示。

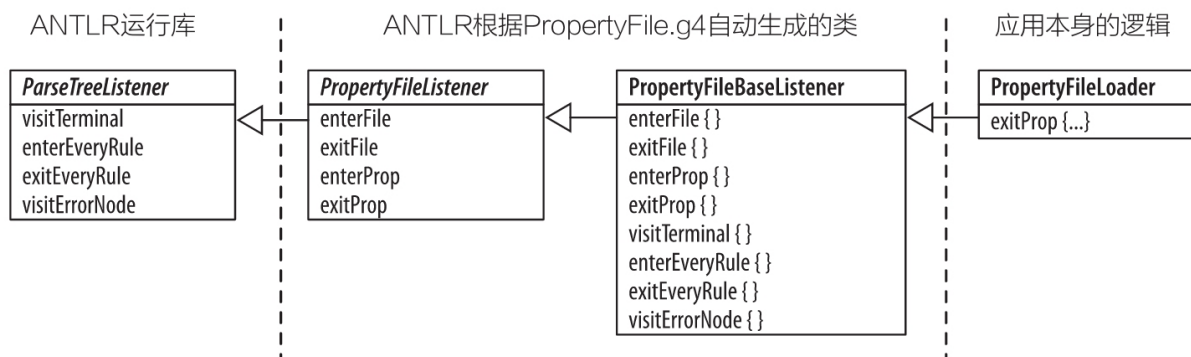


图7-2 接口和类之间的继承关系

`ParseTreeListener`接口位于ANTLR运行库中，它指示每个监听器响应事件`visitTerminal()`、`enterEveryRule()`、`exitEveryRule()`以及（在遇到语法错误时）`visitErrorNode()`。ANTLR根据语法`PropertyFile`自动生成了接口`PropertyFileListener`及其默认实现类`PropertyFileBaseListener`。我们唯一需要编写的是`PropertyFileLoader`，它继承了`PropertyFileBaseListener`中的所有的空方法。

`exitProp()`能够访问该规则的上下文对象，即与`prop`规则相关的`PropContext`。在该上下文对象中，`prop`规则中的每个元素（ID和STRING）都对应一个方法。在语法中，这些元素都是词法符号的引用，因此对应方法返回`TerminalNode`类型的语法分析树节点。我们可以直接通过`getText()`方法获得这些词法符号的文本内容，也可以通过`getSymbol()`方法获得`Token`类型的内容。

下面是最激动人心的成果部分。让我们遍历语法分析树，并在这个过程中使用新的PropertyFileLoader类监听相应的事件。

```
listeners/TestPropertyFile.java
```

```
// 新建一个标准的 ANTLR 语法分析树遍历器
ParseTreeWalker walker = new ParseTreeWalker();
// 新建一个监听器，将其传递给遍历器
PropertyFileLoader loader = new PropertyFileLoader();
walker.walk(loader, tree);          // 遍历语法分析树
System.out.println(loader.props); // 打印结果
```

下列命令复习了对语法运行ANTLR，编译生成的代码，以及启动测试程序处理输入文件的步骤：

```
$ antlr4 PropertyFile.g4
$ ls PropertyFile*.java
PropertyFileBaseListener.java  PropertyFileListener.java
PropertyFileLexer.java        PropertyFileParser.java
$ javac TestPropertyFile.java PropertyFile*.java
$ cat t.properties
user="parrt"
machine="maniac"
$ java TestPropertyFile t.properties
{user="parrt", machine="maniac"}
```

以上结果显示，我们的测试程序成功地将文件中的属性读取到了内存里的Map中。

这种基于监听器的方法十分巧妙，因为所有的遍历过程和方法触发都是自动进行的。有些时候，自动进行的遍历反而成为一个缺陷，因为我们无法控制遍历的过程。例如，我们可能希望遍历一个C语言程序的语法分析树，跳过对代表函数体的子树的访问，从而达到忽略函数内

容的目的。此外，监听器的事件方法也无法利用方法的返回值来传递数据。当需要控制遍历过程，或者希望事件方法返回值时，我们可以使用访问者模式。接下来，作为对比，我们将会构建一个基于访问器机制的属性文件加载器。

7.3 使用访问器编写程序

我们使用访问器机制代替监听器机制的详细步骤是：令ANTLR生成一个访问器接口，实现该接口，然后编写一个测试程序对语法分析树调用visit（）方法。因此，我们完全不需要跟语法交互。

当在命令行中使用“-visitor”选项时，ANTLR自动生成了接口

PropertyFileVisitor和以下默认实现类PropertyFileBaseVisitor:

```
public class PropertyFileBaseVisitor<T> extends AbstractParseTreeVisitor<T>
    implements PropertyFileVisitor<T>
{
    @Override public T visitFile(PropertyFileParser.FileContext ctx) { ... }
    @Override public T visitProp(PropertyFileParser.PropContext ctx) { ... }
}
```

我们可以从上一节的监听器中拷贝exitProp（）中的代码，将它们粘贴到prop规则对应的访问器方法中。

```
listeners/TestPropertyFileVisitor.java
public static class PropertyFileVisitor extends
    PropertyFileBaseVisitor<Void>
{
    Map<String,String> props = new OrderedHashMap<String, String>();
```



```
listeners/TestPropertyFileVisitor.java
```

```
PropertyFileVisitor loader = new PropertyFileVisitor();  
loader.visit(tree);  
System.out.println(loader.props); // 打印结果
```

下面是执行构建和测试的相关命令：

```
$ antlr4 -visitor PropertyFile.g4 # 使用 visitor 选项来创建访问器  
$ ls PropertyFile*.java  
PropertyFileBaseListener.java    PropertyFileListener.java  
PropertyFileBaseVisitor.java     PropertyFileParser.java  
PropertyFileLexer.java           PropertyFileVisitor.java  
$ javac TestPropertyFileVisitor.java  
$ cat t.properties  
user="parrrt"  
machine="maniac"  
$ java TestPropertyFileVisitor t.properties  
{user="parrrt", machine="maniac"}
```

使用访问器和监听器机制，我们可以完成一切与语法相关的事情。一旦进入Java的领域，就没有什么ANTLR的相关内容值得学习了。我们需要谨记在心的是，语法及其对应的语法分析树，以及访问器或者监听器事件方法之间的关系。除此之外，剩下的仅仅是普通的代码。在对输入文本进行识别时，我们可以产生输出、收集信息（正如本例中我们所做的）、用某种方式验证输入文本，或者执行计算。

上文中属性文件的例子非常简单，因而我们无须处理备选分支。默认情况下，ANTLR为每条规则生成单一类型的事件，无论语法分析器匹配到的是其中的哪一个备选分支，该事件都会被触发。这是一件非常不便的事情，因为监听器和访问器方法必须搞清楚语法分析器匹配到

的是哪一个备选分支。在下一节中，我们将会看到如何在更合适的粒度上处理事件。

7.4 标记备选分支以获取精确的事件方法

为了说明过粗的事件粒度带来的问题，让我们尝试利用下列表达式语法生成的监听器构建一个简单的计算器程序。

```
listeners/Expr.g4
grammar Expr;
s : e ;
e : e op=MULT e    // MULT 是 '*'
  | e op=ADD e     // ADD 是 '+'
  | INT
  ;
```

实际上，规则e产生了一个相当鸡肋的监听器方法，因为规则e的所有备选分支都会被遍历器触发为完全相同的enterE () 和exitE () 方法。

```
public interface ExprListener extends ParseTreeListener {
    void enterE(ExprParser.EContext ctx);
    void exitE(ExprParser.EContext ctx);
    ...
}
```

因此，我们实现的监听器方法就不得不做这样的测试：用op词法符号和ctx的方法来判断语法分析器匹配到子树e是哪一个备选分支。

listeners/TestEvaluator.java

```
public void exitE(ExprParser.EContext ctx) {
    if ( ctx.getChildCount()==3 ) { // 本次操作有三个元素
        int left = values.get(ctx.e(0));
        int right = values.get(ctx.e(1));
        if ( ctx.op.getType()==ExprParser.MULT ) {
            values.put(ctx, left * right);
        }
        else {
            values.put(ctx, left + right);
        }
    }
    else {
        values.put(ctx, values.get(ctx.getChild(0))); // 一个 INT
    }
}
```

其中，exitE () 引用的MULT字段是由ANTLR在ExprParser中生成的：

```
public class ExprParser extends Parser {
    public static final int MULT=1, ADD=2, INT=3, WS=4;
    ...
}
```

查看一下ExprParser中的内部类EContext，我们可以发现，ANTLR将这三个备选分支放进了同一个上下文对象中。

```
public static class EContext extends ParserRuleContext {
    public Token op; // 规则中的标签 op
    public List<EContext> e() { ... } // 获取所有的 e 子树
    public EContext e(int i) { ... } // 获取第 i 个 e 子树
    public TerminalNode INT() { ... } // 如果是 e 的第三个备选分支，获取 INT 节点
    ...
}
```

为获取更加精确的监听器事件，ANTLR允许我们用#运算符为任意规则的最外层备选分支提供标签。利用这种方法，我们在Expr语法的基础上，为e的备选分支增加标签，得到LExpr语法：

```
listeners/LExpr.g4
e : e MULT e      # Mult
  | e ADD e       # Add
  | INT           # Int
  ;
```

现在，ANTLR为e的每个备选分支都生成了一个单独的监听器方法，因此，我们不再需要op这个词法符号标签了。对于标签为X的备选分支，ANTLR会自动生成enterX（）和exitX（）。

```
public interface LExprListener extends ParseTreeListener {
    void enterMult(LExprParser.MultContext ctx);
    void exitMult(LExprParser.MultContext ctx);
    void enterAdd(LExprParser.AddContext ctx);
    void exitAdd(LExprParser.AddContext ctx);
    void enterInt(LExprParser.IntContext ctx);
    void exitInt(LExprParser.IntContext ctx);
    ...
}
```

需要注意的是，ANTLR也为不同的备选分支生成了特定的上下文对象（EContext的子类），并以标签命名。这样的上下文对象中的getter方法被限制为只能获取对应备选分支中的内容。例如，IntContext只有一个INT（）方法。我们可以在enterInt（）中调用ctx.INT（），但是不能在enterAdd（）中调用它。

监听器和访问器方法是非常精彩的设计。通过在事件方法中实现具体逻辑，我们成功地在封装语言类应用程序的同时，获得了可被复用的语法。ANTLR甚至帮我们生成了代码的框架。尽管如此，直到现在，我们编写的程序还是过于简单，以至于在实现的过程中，我们都没有

遇到这样一个常见问题：有些时候，事件方法需要传递局部调用的结果或者其他信息。

7.5 在事件方法中共享信息

不论是出于收集信息还是计算的目的，良好的编程实践应该使用传参和返回值，而非类成员或者其他的“全局变量”。但是，问题在于，ANTLR自动生成的监听器方法是不带自定义返回值和参数的。同样，ANTLR生成的访问器方法也不带自定义参数。

在本节中，我们将会研究在不改变事件方法签名的前提下，用它们来传递数据的机制。我们将会使用三种方法实现上一节提到的，基于LExpr表达式语法的示例计算器程序。其中，第一种方法使用访问器方法来返回值，第二种使用类成员在事件方法之间共享数据，第三种通过对语法分析树的节点进行标注来存储相关数据。

1.使用访问器遍历语法分析树

为构建一个基于访问器的计算器程序，最简单的方法是令expr规则中的事件方法返回子表达式的值。例如，visitAdd（）将返回两个子表达式相加的结果，visitInt（）方法将返回整数元素的值。传统的访问器通常不指定方法的返回值。为这些访问器方法增加返回值类型十分容易：在实现自定义的程序逻辑的时候，继承LExprBaseVisitor<T>并指定Integer作为泛型<T>参数即可。我们实现的访问器如下所示：

listeners/TestLEvalVisitor.java

```
public static class EvalVisitor extends LExprBaseVisitor<Integer> {
    public Integer visitMult(LExprParser.MultContext ctx) {
        return visit(ctx.e(0)) * visit(ctx.e(1));
    }

    public Integer visitAdd(LExprParser.AddContext ctx) {
        return visit(ctx.e(0)) + visit(ctx.e(1));
    }

    public Integer visitInt(LExprParser.IntContext ctx) {
        return Integer.valueOf(ctx.INT().getText());
    }
}
```

EvalVisitor从ANTLR的AbstractParseTreeVisitor类继承了通用的visit()
()，我们将它作为触发子树访问的捷径。

注意EvalVisitor中没有s规则对应的访问器方法。在LExprBaseVistor中，visitS () 的默认实现会调用预定义的ParseTreeVisitor.visitChildren () 方法。visitChildren () 方法返回访问最后一个子节点的返回值。在本例中，visitS () 返回对它的唯一子节点（即e节点）进行访问得到的返回值。因此，我们可以利用这种默认行为。

在测试代码TestLEvalVisitor.java中，我们照常启动了LExprParser并打印出了语法分析树。接下来，我们需要一些代码启动EvalVisitor，并在遍历完该树后打印出表达式的计算结果。

listeners/TestLEvalVisitor.java

```
EvalVisitor evalVisitor = new EvalVisitor();
int result = evalVisitor.visit(tree);
System.out.println("visitor result = "+result);
```

为完成计算器程序的构建，和之前章节相同，我们使用“-visitor”选项来通知ANTLR生成访问器（如果我们不希望ANTLR生成监听器，可以使用选项“-no-listener”）。下面是完整的构建和测试步骤：

```
⇒ $ antlr4 -visitor LExpr.g4
⇒ $ javac LExpr*.java TestLEvalVisitor.java
⇒ $ java TestLEvalVisitor
⇒ 1+2*3
⇒ EOf
  ( (s (e (e 1) + (e (e 2) * (e 3)))) )
  visitor result = 7
```

如果我们需要让自定义的程序返回值，访问器是个不错的选择，因为它使用的是Java原生的返回值机制。如果我们不希望每次都显式调用访问器方法来访问子节点，我们可以换成监听器机制。然而不幸的是，这意味着我们放弃了使用Java方法返回值带来的整洁。

2.使用栈来模拟返回值

ANTLR生成的监听器方法是没有返回值的（void类型）。在语法分析树中，为了向监听器方法的更高层的调用者返回值，我们可以将监听器的局部结果保存在一个成员变量中。首先浮现在我们脑海里的是栈结构，就像Java虚拟机使用栈来临时存储返回值那样。即，每个子表达式的计算结果推入一个栈中。下面是完整的Evaluator计算程序的监听器的代码（位于文件TestLEvaluator.java中）：

listeners/TestLEvaluator.java

```
public static class Evaluator extends LExprBaseListener {
    Stack<Integer> stack = new Stack<Integer>();

    public void exitMult(LExprParser.MultContext ctx) {
        int right = stack.pop();
        int left = stack.pop();
        stack.push( left * right );
    }

    public void exitAdd(LExprParser.AddContext ctx) {
        int right = stack.pop();
        int left = stack.pop();
        stack.push(left + right);
    }

    public void exitInt(LExprParser.IntContext ctx) {
        stack.push( Integer.valueOf(ctx.INT().getText()) );
    }
}
```

我们可以按照在之前章节的TestPropertyFile中做的那样，新建并使用一个ParseTree-Walker，配合TestLEvaluator来验证这种方法的正确性。

```
⇒ $ antlr4 LExpr.g4
⇒ $ javac LExpr*.java TestLEvaluator.java
⇒ $ java TestLEvaluator
⇒ 1+2*3
⇒ E0F
  ( ( s ( e ( e 1 ) + ( e ( e 2 ) * ( e 3 ) ) ) ) )
    stack result = 7
```

这种使用栈的方法不够优雅，但是非常有效。通过它，我们可以保证事件方法在所有的监听器事件之间的执行顺序是正确的。带返回值的访问器足够优雅，但是需要我们手工触发对树节点的访问。此外，第三种方案是把局部结果保存在树节点里。

3.标注（Annotate）语法分析树

在上文中，我们使用了临时存储在事件方法之间共享数据。作为一个替代方案，我们可以将这些数据直接存储在语法分析树里。监听器和访问器机制都支持树的标注，接下来我们会用监听器进行演示。首先，让我们看看 $1+2*3$ 生成的带局部结果标注的LExpr语法分析树，如图7-4所示。

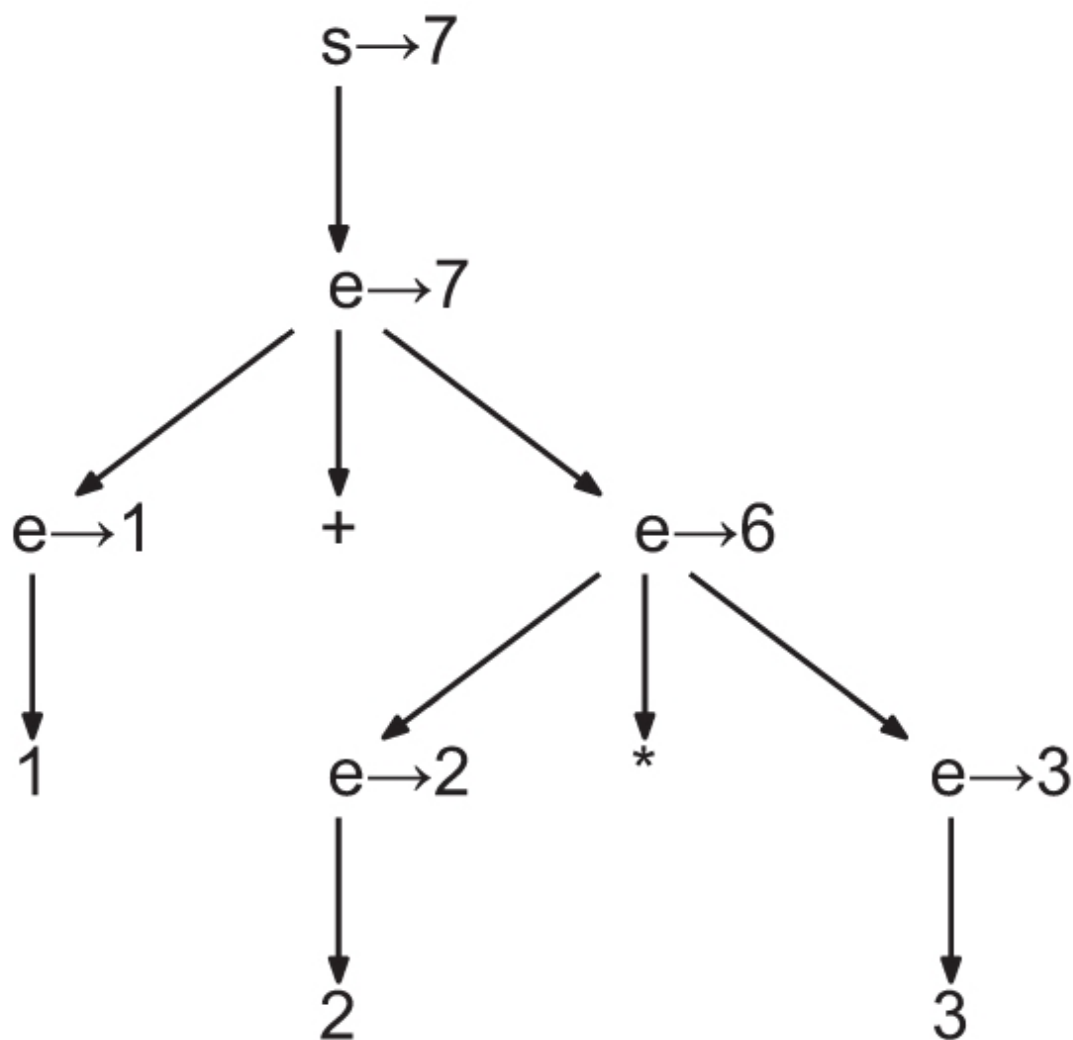


图7-4 LExpr语法分析树

其中，每个子表达式对应一个子树的根节点（同时对应一个`e`规则的调用）。每个由`e`节点开始的水平箭头指向的数字就是我们希望“返回”的局部计算结果。

让我们看看在`LExpr`语法中，节点标注是怎样使规则`e`正常工作的。

```
listeners/LExpr.g4
e : e MULT e      # Mult
  | e ADD e       # Add
  | INT           # Int
  ;
```

`e`的备选分支对应的监听器方法将会在对应的语法分析树节点中保存一个结果。在更高层的节点上，后续的任何加法或者乘法事件触发的方法都可以通过查找对应的子节点中存储的值来获得子表达式的结果。

通过规则参数和返回值为节点添加字段

如果我们不在乎将语法绑定在特定的编程语言上所引起灵活性丧失，我们可以简单地为特定规则添加一个返回值。

```
e returns [int value]
: e '*' e      # Mult
| e '+' e      # Add
| INT         # Int
;
```

ANTLR会将所有的参数和返回值放入相关的上下文对象中，这样，`value`就成为`EContext`的一个字段。


```
public static class EContext extends ParserRuleContext {  
    public int value;  
    ...  
}
```

因为相应备选分支中的上下文类都继承自**EContext**，所有的监听器方法都能访问这个值。例如，监听器方法可以直接使用**ctx.value=0**。

这里展示的这种方法指定某条规则产生一个结果值，该值将被存储于此规则的上下文对象内。此过程使用了目标语言的片段，从而导致这个语法被绑定到了特定的目标语言上。不过，这种方法并不意味着这份语法被绑定到了特定的应用程序上，因为其他程序可能也需要此规则产生同样的结果值。

我们暂且假设语法分析树的每个节点（每个规则的上下文对象）都有一个名为**value**的字段，那么**exitAdd ()**方法就会像下面一样：

```
public void exitAdd(LExprParser.AddContext ctx) {  
    // e(0).value 是备选分支中的第一个 e 子表达式的值  
    ctx.value = ctx.e(0).value + ctx.e(1).value; // e '+' e # Add  
}
```

这种方法看似合理，然而不幸的是，在**Java**中，我们无法为**ExprContext**类动态添加一个**value**字段（像**Ruby**和**Python**那样）。为了使语法分析树的标注生效，我们需要一种标注多个节点的方法，这种方法不能是手工修改**ANTLR**生成的相关节点类，因为**ANTLR**下次生成代码时会覆盖掉我们的修改。

最简单的标注语法分析树节点的方法是使用**Map**来将任意值和节点一一对应起来。出于这个目的，**ANTLR**提供了一个简单的名为**ParseTreeProperty**的辅助类。让我们构建另外一个版本的计算器程序，称为**EvaluatorWithProps**。它位于文件**TestLEvaluatorWithProps.java**中，通过一个**ParseTreeProperty**实例将局部计算结果和**LExpr**语法分析树节点关联起来。下面是我们实现的监听器的开头部分：

```
listeners/TestLEvaluatorWithProps.java
```

```
public static class EvaluatorWithProps extends LExprBaseListener {  
    /** 使用 Map<ParseTree,Integer> 将节点映射到对应的整数值 */  
    ParseTreeProperty<Integer> values = new ParseTreeProperty<Integer>();
```

千万小心，如果你想要使用自己的**Map**来代替**ParseTreeProperty**，确保它是从**IdentityHashMap**而非常规的**HashMap**派生的。我们使用同一性而非**equals()**来标注特定的节点并完成测试。两个**e**节点可能是**equals()**的，但却并非内存中的同一个节点实例。

我们可以使用**values.put(node, value)**来标注一个节点，使用**values.get(node)**来获取一个节点对应的值。这是完全可行的，不过最好编写一些清晰明了的辅助方法来增强代码的可读性。

```
listeners/TestLEvaluatorWithProps.java
```

```
public void setValue(ParseTree node, int value) { values.put(node, value); }  
public int getValue(ParseTree node) { return values.get(node); }
```

让我们从最简单的表达式备选分支——**Int**——开始编写监听器方法。我们希望用它匹配的**INT**词法符号的值来标注它对应的语法分析树节点

e °

```
listeners/TestLEvaluatorWithProps.java
public void exitInt(LExprParser.IntContext ctx) {
    String intText = ctx.INT().getText(); // INT # Int
    setValue(ctx, Integer.valueOf(intText));
}
```

对于加法子树，我们首先获取它的两个子表达式（操作数）的值，然后用二者的和来标注该子树的根节点。

```
listeners/TestLEvaluatorWithProps.java
public void exitAdd(LExprParser.AddContext ctx) {
    int left = getValue(ctx.e(0)); // e '+' e # Add
    int right = getValue(ctx.e(1));
    setValue(ctx, left + right);
}
```

`exitMult ()` 方法和上面相似，只是将加法换成了乘法。

我们的测试程序的起始规则是`s`，因此我们必须确保语法分析树的根节点包含子树`e`（或者，我们也可以把`e`作为起始的解析规则）。为了让结果值从`e`节点向`s`节点“冒泡”，我们实现了`exitS ()` 方法。

```
listeners/TestLEvaluatorWithProps.java
/** Need to pass e's value out of rule s : e ; */
public void exitS(LExprParser.SContext ctx) {
    setValue(ctx, getValue(ctx.e())); // 类比：int s() { return e(); }
}
```

下面是启动监听器和打印根节点对应的表达式值的代码：

```
listeners/TestLEvaluatorWithProps.java
```

```
ParseTreeWalker walker = new ParseTreeWalker();  
EvaluatorWithProps evalProp = new EvaluatorWithProps();  
walker.walk(evalProp, tree);  
System.out.println("properties result = " +evalProp.getValue(tree));
```

下面是构建和测试的步骤:

```
⇒ $ antlr4 LExpr.g4  
⇒ $ javac LExpr*.java TestLEvaluatorWithProps.java  
⇒ $ java TestLEvaluatorWithProps  
⇒ 1+2*3  
⇒ E0  
  ⚡ (s (e (e 1) + (e (e 2) * (e 3))))  
    properties result = 7
```

现在，我们拥有了计算器程序的三个不同实现，也准备好将所学的知识应用于构建真实的例子了。由于每种方法都有优缺点，让我们来比较这些不同的方法，同时回顾一下我们在本章中的收获。

4.不同的数据共享方法对比

为获取可复用的语法，我们需要使其与用户自定义的动作分离。这意味着将所有程序自身的逻辑代码放到语法之外的某种监听器或者访问器中。监听器和访问器通过操纵语法分析树来完成工作，ANTLR会自动生成合适的接口和默认实现类，以便对语法分析树进行遍历。但是，由于事件方法的签名是固定的，无法由程序自行决定，我们找到了三种在事件方法间共享数据的方案。

- 原生的Java调用栈：访问器返回一个用户指定类型的值。不过，如果访问器需要传递参数，那就必须使用下面两种方案。

- 基于栈的：在上下文类中维护一个栈字段，以与Java调用栈相同的方式，模拟参数和返回值的入栈和出栈。

- 标注：在上下文类中维护一个Map字段，用对应的值来标注节点。

这三种方案都能将程序的具体逻辑封装在特定的对象内，从而使其与语法本身完全解耦。除此之外，它们各有利弊。我们需要综合考虑实际问题以及个人喜好，来决定使用哪种方案。当然，我们也可以在同一个程序中同时使用多种解决方案。

使用访问器方法的代码具有良好的可读性，这是因为它们直接调用其他的访问器方法来获得局部计算结果，同时能像其他方法一样返回值。这一点既是优点也是缺点。访问器方法必须显式访问它们的子节点，而监听器无须如此。因为访问器的接口是通用的，因此在其中无法使用自定义的参数。访问器必须使用其他两种方案之一来解决调用子节点的访问器方法时的传参问题。访问器方法的空间效率较高，因为它在某一时刻只需保存少量的局部结果。在完成对树的遍历之后，局部结果就不存在了。虽然访问器方法能够返回值，但是所有的值都必须具有相同的类型，其他方案不会受到这样的限制。

基于栈的解决方案能够使用栈来模拟参数和返回值，但是，在手工操纵栈的过程中，存在失误的可能性。这种情况可能在监听器方法没有直接调用其他的监听器方法时发生。作为开发者，我们必须确保，在未来的事件方法调用中，推入栈中的内容被正确地弹出。栈可以传递多个参数值和多个返回值。基于栈的解决方案同样具有较高的空间效率，因为它们不会在树中存储任何东西。所有局部结果的存储在树遍历完成后都会被释放。

树标注是我个人的首选解决方案，因为它允许我向事件方法提供任意信息来操纵语法分析树中的各个节点。通过该方案，我可以传递多个任意类型的参数值。在很多情况下，标注比存储转瞬即逝的值的栈更好。使用它，在众多方法中来回传递数据也更不容易失误。这种方案的唯一缺点是，在整个遍历的过程中，局部结果都会被保留，因此具有更大的内存消耗。

另一方面，某些程序恰好需要标注语法分析树的方案，例如8.4节。该程序需要对语法分析树进行多次遍历，将第一趟遍历得到的数据完整地储存在树中是合理的，这样，第二趟遍历就能非常容易地获取这些数据。总之，对树进行标注的方案异常灵活，同时内存占用也处于可接受的范围。

现在，我们知道了如何使用树监听器和访问器来实现基本的语言类应用程序，是时候基于这些技术来构建一些真正的工具程序了。这就是

我们下一章所要完成的工作。

第8章 构建真实的语言类应用程序

在之前章节的学习中，我们已经掌握了通过监听器和访问器来调用自定义程序的方法，因此，编写一些实用程序的时机已经成熟。我们将会基于第6章中完成的CSV、JSON和Cymbol语法，由浅入深地构造四个监听器（同样，我们也可以使用访问器，它同样简单）。

第一个实用程序是读取CSV文件的加载器，这样的CSV文件可以看作一种二维列表的数据结构。接着，我们将会解决将JSON文本翻译成XML文件的问题。之后，我们将会读取Cymbol程序，使用DOT/graphviz将其函数调用依赖图可视化。最后，我们会为Cymbol构造一个真实的符号表，用于检测未定义的变量和函数，确保变量和函数被正确运用。验证器需要对语法分析树进行多趟扫描，因此，我们可以用这个例子来展示，如何在一趟扫描中收集数据，并在后续过程中使用。

下面让我们从最简单的程序开始。

8.1 加载CSV数据

我们的目标是编写一个自定义的监听器，将逗号分隔符文件（CSV）中的数据加载到一种精心设计的数据结构——“由Map组成的List”中。这是一件其他数据读取器甚至一个配置文件读取器都能够完成的事情。我们会为每个行建立一个Map，其中包含从列名到字段的映射。因此，对于如下输入文件：

```
listeners/t.csv
Details,Month,Amount
Mid Bonus,June,"$2,000"
,January,"""zippo"""
Total Bonuses,"","$5,000"
```

我们预期的“由Map组成的List”如下所示：

```
[{Details=Mid Bonus, Month=June, Amount="$2,000"},
 {Details=, Month=January, Amount="""zippo"""},
 {Details=Total Bonuses, Month="", Amount="$5,000"}]
```

为获得更精确的监听器方法，让我们对6.1节中完成的CSV语法的备选分支进行标记。


```
listeners/CSV.g4
```

```
grammar CSV;

file : hdr row+ ;
hdr : row ;

row : field (',' field)* '\r'? '\n' ;

field
    :   TEXT      # text
    |   STRING    # string
    |           # empty
    ;

TEXT : ~[,\n\r"]+ ;
STRING : '"' ( '"' | ~'"' )* '"' ;
```

除此之外，这个CSV语法和之前的版本相同。

我们可以从定义所需的数据结构开始，逐步完成这个监听器的实现。

首先，我们需要一个核心数据结构rows，它是一种由Map组成的List。

其次，我们还需要一组列名，它来源于标题行header。在对一行数据的处理过程中，我们会将其所有字段值放入一个临时列表

currentRowFieldValues中，并在最终完成对该行处理的时候，将列名映射到这些字段值上。

下面就是我们的监听器的开始部分：

```
listeners/LoadCSV.java
```

```
public static class Loader extends CSVBaseListener {
    public static final String EMPTY = "";
    /** 这个列表中的每个元素是一个代表一行数据的 Map，该 Map 是从字段名到字段值的映射 */
    List<Map<String,String>> rows = new ArrayList<Map<String, String>>();
    /** 列名的列表 */
    List<String> header;
    /** 构造一个存放当前行中所有字段值的列表 */
    List<String> currentRowFieldValues;
```

下列三个规则方法处理字段值的方式是：提取合适的字符串，并将其加入currentRowFieldValues。

listeners/LoadCSV.java

```
public void exitString(CSVParser.StringContext ctx) {
    currentRowFieldValues.add(ctx.STRING().getText());
}

public void exitText(CSVParser.TextContext ctx) {
    currentRowFieldValues.add(ctx.TEXT().getText());
}

public void exitEmpty(CSVParser.EmptyContext ctx) {
    currentRowFieldValues.add(EMPTY);
}
```

在处理每行数据之前，我们需要从首行中获取全部列名。虽然标题行在语法上只是一个普通的行，但是我们需要将它和普通的数据行区别对待。这意味着需要检查上下文。我们不妨暂时假设在首行的exitRow

()方法执行结束后，currentRowFieldValues中就包含了全部的列名。我们只需要从其中提取字段值即可填充header。

listeners/LoadCSV.java

```
public void exitHdr(CSVParser.HdrContext ctx) {
    header = new ArrayList<String>();
    header.addAll(currentRowFieldValues);
}
```

接着我们回过头来处理行数据，这个过程需要两个操作：开始和结束对一行的处理。当开始对一行的处理时，我们需要创建（或者清除）currentRowFieldValues，以备接收后续数据。

```
listeners/LoadCSV.java
```

```
public void enterRow(CSVParser.RowContext ctx) {  
    currentRowFieldValues = new ArrayList<String>();  
}
```

在完成对一行的处理时，我们需要考虑上下文。如果我们刚刚处理的行是标题行，那就什么都不做，因为列名不是数据。在`exitRow()`方法中，我们可以通过查看父节点的`getRuleIndex()`的返回值得知当前的上下文（或者查看父节点的类型是否是`HdrContext`）。如果当前行是一个数据行，我们就创建一个`Map`，同步遍历`header`中的列名和`currentRow-FieldValues`中的字段值，并将映射关系放入该`Map`中。

```
listeners/LoadCSV.java
```

```
public void exitRow(CSVParser.RowContext ctx) {  
    // 如果当前行是标题行，什么都不做  
    // if ( ctx.parent instanceof CSVParser.HdrContext ) return; 或者:  
    if ( ctx.getParent().getRuleIndex() == CSVParser.RULE_hdr ) return;  
    // 当前行是数据行  
    Map<String, String> m = new LinkedHashMap<String, String>();  
    int i = 0;  
    for (String v : currentRowFieldValues) {  
        m.put(header.get(i), v);  
        i++;  
    }  
    rows.add(m);  
}
```

这就是将CSV数据读入精心设计的数据结构所需的全部工作。在使用一个`ParseTree-Walker`完成对语法分析树的遍历后，我们的`LoadCSV`类中的`main()`方法就能打印出包含全部数据的`rows`。

```
listeners/LoadCSV.java
```

```
ParseTreeWalker walker = new ParseTreeWalker();  
Loader loader = new Loader();  
walker.walk(loader, tree);  
System.out.println(loader.rows);
```

下面是构建和测试的步骤:

```
$ antlr4 CSV.g4  
$ javac CSV*.java LoadCSV.java  
$ java LoadCSV t.csv  
[{"Details=Mid Bonus, Month=June, Amount="$2,000"}, {"Details=, Month=January,  
Amount=""zippo""}, {"Details=Total Bonuses, Month="", Amount="$5,000"}]
```

在读取数据之后, 我们可能希望将其翻译为另外一种语言——这也是我们下一节所要研究的内容。

8.2 将JSON翻译成XML

许多网络服务返回JSON数据，有时候，我们希望将一些JSON数据输入某个只接受XML的程序。让我们以6.2节中得到的JSON语法为基础，构建一个从JSON到XML的翻译器。我们的目标是读取这样的输入：

```
listeners/t.json
{
  "description" : "An imaginary server config file",
  "logs" : {"level":"verbose", "dir":"/var/log"},
  "host" : "antlr.org",
  "admin": ["parrt", "tombu"],
  "aliases": []
}
```

并给出等价的XML输出：

```
<description>An imaginary server config file</description>
<logs>
  <level>verbose</level>
  <dir>/var/log</dir>
</logs>
<host>antlr.org</host>
<admin>
  <element>parrt</element>
  <element>tombu</element>
</admin>
<aliases></aliases>
```

其中，<element>是一个我们需要在翻译过程中生成的标签。

和CSV语法一样，让我们首先对JSON语法中的备选分支做一定的标记，以便ANTLR生成更精确的监听器方法。

```
listeners/JSON.g4
```

```
object
:   '{' pair (',' pair)* '}'   # AnObject
|   '{' '}'                   # EmptyObject
;

array
:   '[' value (',' value)* ']' # ArrayOfValues
|   '[' ']'                   # EmptyArray
;
```

我们会用同样的方法处理`value`规则，不过稍微做出了一些改变。除了其中三个备选分支之外，其他都必须返回它匹配到的文本值，因此，我们可以对它们进行同样的标记，使得语法分析树遍历器为这些备选分支触发相同的监听器事件。

```
listeners/JSON.g4
```

```
value
:   STRING           # String
|   NUMBER           # Atom
|   object           # ObjectValue
|   array            # ArrayValue
|   'true'           # Atom
|   'false'          # Atom
|   'null'           # Atom
;
```

我们的翻译器的实现需要令每条规则返回与它匹配到的输入文本等价的XML。为跟踪这些局部结果，我们将会使用`xml`字段和两个辅助方法来对语法分析树进行标注。

```
listeners/JSON2XML.java
```

```
public static class XMLEmitter extends JSONBaseListener {
    ParseTreeProperty<String> xml = new ParseTreeProperty<String>();
    String getXML(ParseTree ctx) { return xml.get(ctx); }
    void setXML(ParseTree ctx, String s) { xml.put(ctx, s); }
```

我们会将每棵子树翻译完的字符串存储在该子树的根节点中。这样，工作在语法分析树更高层节点上的方法就能够获得它们，从而构造出更大的字符串。语法分析树的根节点中存储的字符串就是最终的翻译结果。

让我们从最简单的翻译开始。**value**规则中的**Atom**备选分支“返回”（也就是在**Atom**节点对应的标注值）它匹配的词法符号中的文本内容（`ctx.getText()` 获得对应规则匹配到的文本）。

```
listeners/JSON2XML.java
public void exitAtom(JSONParser.AtomContext ctx) {
    setXML(ctx, ctx.getText());
}
```

除了需要额外剥离双引号之外（`stripQuotes()` 是该文件提供的辅助方法），对字符串的处理基本和上述过程基本相同。

```
listeners/JSON2XML.java
public void exitString(JSONParser.StringContext ctx) {
    setXML(ctx, stripQuotes(ctx.getText()));
}
```

如果**rule()**方法匹配到的是一个对象或者一个数组，它就可以将这些复合元素的翻译结果拷贝到自身的语法分析树节点中，下列代码给出了实现细节：

```
listeners/JSON2XML.java
public void exitObjectValue(JSONParser.ObjectValueContext ctx) {
    // 类比 String value() {return object();}
    setXML(ctx, getXML(ctx.object()));
}
```

在翻译完成value规则对应的所有元素后，我们需要处理键值对，将它们转换为标签和文本。生成的XML的标签名来源于STRING：'value'备选分支中的STRING。开始和结束标签之间的文本来源于value子节点。

listeners/JSON2XML.java

```
public void exitPair(JSONParser.PairContext ctx) {
    String tag = stripQuotes(ctx.STRING().getText());
    JSONParser.ValueContext vctx = ctx.value();
    String x = String.format("<%s>%s</%s>\n", tag, getXML(vctx), tag);
    setXML(ctx, x);
}
```

JSON的对象由一系列键值对组成。因此，对于每个object规则在AnObject备选分支中发现的键值对，我们将其对应的XML追加到语法分析树中储存的结果之后。

listeners/JSON2XML.java

```
public void exitAnObject(JSONParser.AnObjectContext ctx) {
    StringBuilder buf = new StringBuilder();
    buf.append("\n");
    for (JSONParser.PairContext pctx : ctx.pair()) {
        buf.append(getXML(pctx));
    }
    setXML(ctx, buf.toString());
}
public void exitEmptyObject(JSONParser.EmptyObjectContext ctx) {
    setXML(ctx, "");
}
}
```

处理数组的方式与之相似，从各子节点中获取XML结果，将其分别放入<element>标签之后连接起来即可。

listeners/JSON2XML.java

```
public void exitArrayOfValues(JSONParser.ArrayOfValuesContext ctx) {
    StringBuilder buf = new StringBuilder();
```



```

        buf.append("\n");
        for (JSONParser.ValueContext vctx : ctx.value()) {
            buf.append("<element>"); // 将所有元素连接成合法的 XML 文本
            buf.append(getXML(vctx));
            buf.append("</element>");
            buf.append("\n");
        }
        setXML(ctx, buf.toString());
    }

    public void exitEmptyArray(JSONParser.EmptyArrayContext ctx) {
        setXML(ctx, "");
    }
}

```

最后，我们需要用最终的结果——由根元素`object`或者`array`生成的结果——标注语法分析树的根节点。

listeners/JSON.g4

```

json:    object
        |    array
        ;

```

我们可以用一个简单的`set`操作来完成这项工作。

listeners/JSON2XML.java

```

public void exitJson(JSONParser.JsonContext ctx) {
    setXML(ctx, getXML(ctx.getChild(0)));
}

```

下面是构建和测试的步骤:

```

$ antlr4 JSON.g4
$ javac JSON*.java
$ java JSON2XML t.json

```

```

<description>An imaginary server config file</description>
<logs>
<level>verbose</level>
...

```

翻译并非总是像JSON转XML一样直来直往。上述例子的意义在于向我们展示了解决翻译问题的方法：分而治之，然后将局部结果合并（如果你查看一下源代码目录，就可以发现另一个版本的代码JSON2XML_ST.java，其中使用了StringTemplate来生成输出，此外，还有一个版本的代码JSON2XML_DOM.java，用于构建XML的DOM树）。

好了，我们已经处理了足够多的数据描述语言，下面让我们对编程语言做一些有趣的事情吧。

8.3 生成调用图

软件的编写和维护并非一帆风顺的，这也是我们试图使用工具来提高生产率和效率的原因。例如，在过去的几十年中，我们见到了测试框架、代码覆盖工具和代码分析工具的爆炸性增长。此外，用可视化的树来检视类继承关系也是一件好事，这是大多数开发环境所支持的。另一种我喜爱的可视化方案称为调用图，其中的节点是函数，节点间的有向边是函数的调用。

在本节中，我们将使用来自6.4节的Cymbol语法编写一个调用图生成器。它的简单会让你大吃一惊，尤其是当你看到它结果的精妙之后。为让你大致了解一下我们试图达成的目标，请看下面的函数和函数调用：

listeners/t.cymbol

```
int main() { fact(); a(); }

float fact(int n) {
    print(n);

    if ( n==0 ) then return 1;
    return n * fact(n-1);
}

void a() { int x = b(); if false then {c(); d();} }
void b() { c(); }
void c() { b(); }
void d() { }
void e() { }
```

我们期望生成如图8-1所示的调用图：

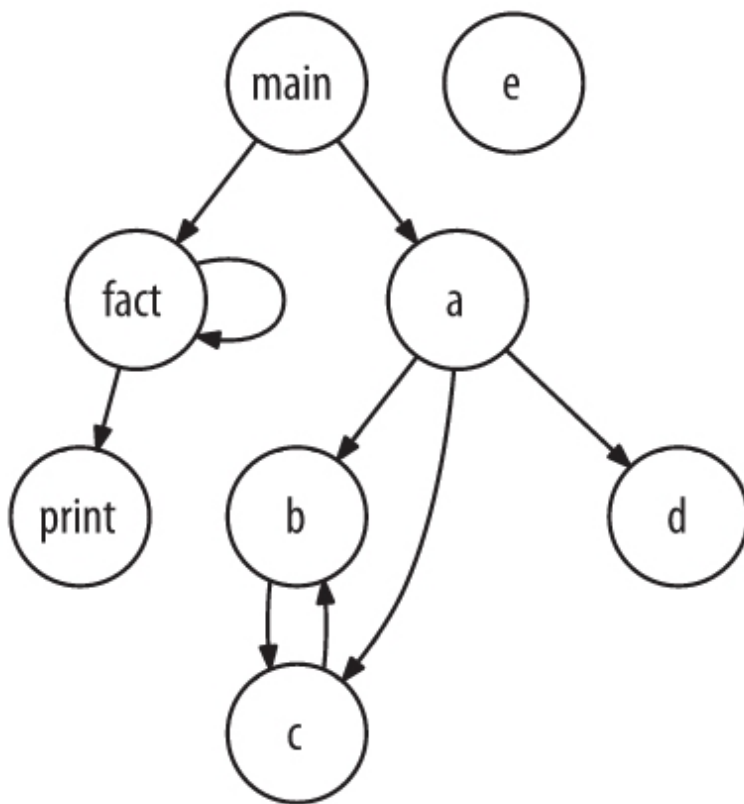


图8-1 期望生成的调用图

可视化展示的优点在于人们能够轻易分辨出其中的异常。例如，`e ()` 节点是一个孤立节点，这意味着它没有被任何函数调用，因此是无用代码（**dead code**）。只需扫一眼，我们就发现了一个可以丢弃的函数。此外，我们还可以轻易通过检查图中的循环来发现递归，例如 `fact () → fact ()` 和 `b () → c () → b ()`。

为了生成这样的可视化调用图，我们需要读取Cymbol程序，根据它产生一个DOT文件（然后利用graphviz进行预览）。例如，我们需要基于之前的例子t.cymbol生成一个如下所示的DOT文件。

```
digraph G {
    ranksep=.25;
    edge [arrowsize=.5]
    node [shape=circle, fontname="ArialNarrow",
          fontsize=12, fixedsize=true, height=.45];
    main; fact; a; b; c; d; e;
    main -> fact;
    main -> a;

    fact -> print;
    fact -> fact;
    a -> b;
    a -> c;
    a -> d;
    b -> c;
    c -> b;
}
```

输出结果包含一个类似“`ranksep=.25;`”的模板语句，其后是一列节点和边。为了捕获孤立节点，我们需要保证为每个函数名生成一个节点定

义，即使在它不与进出边相连接的情况下。如果不对这种情况进行特殊处理，这样的节点将不会出现在图中。注意其中节点定义行末尾的e节点。

```
main; fact; a; b; c; d; e;
```

我们的策略相当直白。当语法分析器遇到函数声明的时候，我们的程序将会把该函数的名字加入一个列表中，然后在一个名为 `currentFunctionName` 的字段中记录它。当语法分析器遇到一个函数调用时，我们的程序将会记录下一条从 `currentFunctionName` 到被调用函数名的边。

首先，我们为 `Cymbol.g4` 中的一些备选分支进行标记，以获取更精确的监听器方法。

listeners/Cymbol.g4

```
expr: ID '(' exprList? ')' # Call
    | expr '[' expr ']' # Index
    | '-' expr # Negate
    | '!' expr # Not
    | expr '*' expr # Mult
    | expr ('+'|'-') expr # AddSub
    | expr '==' expr # Equal
    | ID # Var
    | INT # Int
    | '(' expr ')' # Parens
    ;
```

之后，让我们将所有与图相关的代码都封装进一个类中，作为我们的语言类应用程序的基础。

listeners/CallGraph.java

```
static class Graph {
    // 这里使用的是 org.antlr.v4.runtime.misc: OrderedHashSet, MultiMap
    Set<String> nodes = new OrderedHashSet<String>(); // 函数的列表
    MultiMap<String, String> edges = // 调用者 -> 被调用者
        new MultiMap<String, String>();
    public void edge(String source, String target) {
        edges.map(source, target);
    }
}
```

有了节点和边的集合，我们就可以在Graph类中编写一个Java的小方法toDOT（）来完整地获取对应的DOT代码。

listeners/CallGraph.java

```
public String toDOT() {
    StringBuilder buf = new StringBuilder();
    buf.append("digraph G {\n");
    buf.append("    ranksep=.25;\n");
    buf.append("    edge [arrowsize=.5]\n");
    buf.append("    node [shape=circle, fontname=\"ArialNarrow\", \n");
    buf.append("        fontsize=12, fixedsize=true, height=.45];\n");
    buf.append(" ");
    for (String node : nodes) { // 首先打印所有节点
        buf.append(node);
        buf.append("; ");
    }
    buf.append("\n");
    for (String src : edges.keySet()) {
        for (String trg : edges.get(src)) {
            buf.append(" ");
            buf.append(src);
            buf.append(" -> ");
            buf.append(trg);
            buf.append(";\n");
        }
    }
    buf.append("}\n");
    return buf.toString();
}
```

现在，我们需要做的一切就是使用监听器填充这些数据结构。该监听器需要两个用于记录的字段。

listeners/CallGraph.java

```
static class FunctionListener extends CymbolBaseListener {  
    Graph graph = new Graph();  
    String currentFunctionName = null;
```

它只需要监听两个方法。第一个是当语法分析器遇到函数定义时的方法，令其记录当前函数名。

listeners/CallGraph.java

```
public void enterFunctionDecl(CymbolParser.FunctionDeclContext ctx) {  
    currentFunctionName = ctx.ID().getText();  
    graph.nodes.add(currentFunctionName);  
}
```

接着，当语法分析器发现函数调用时，程序就会记录一条从当前函数到被调用的函数的边。

listeners/CallGraph.java

```
public void exitCall(CymbolParser.CallContext ctx) {  
    String funcName = ctx.ID().getText();  
    // 将当前函数映射到被调用函数上  
    graph.edge(currentFunctionName, funcName);  
}
```

需要注意的是，函数调用不能出现在嵌套的代码块或者声明中，如下面代码中的a（）。

```
void a() { int x = b(); if false then {c(); d();} }
```

无论语法分析树遍历器在何处遇到函数调用，它都会触发exitCall（）监听器方法。

通过语法分析树和上面的FunctionListener类，我们就可以用我们在遍历中使用我们自定义的监听器，并产生我们期望的输出。

```
listeners/CallGraph.java
```

```
ParseTreeWalker walker = new ParseTreeWalker();
FunctionListener collector = new FunctionListener();
walker.walk(collector, tree);
System.out.println(collector.graph.toString());
System.out.println(collector.graph.toDOT());
```

在输出DOT代码之前，上述代码会打印出函数名和边的列表。

```
$ antlr4 Cymbol.g4
$ javac Cymbol*.java CallGraph.java
$ java CallGraph t.cymbol
edges: {main=[fact, a], fact=[print, fact], a=[b, c, d], b=[c], c=[b]},
functions: [main, fact, a, b, c, d, e]
digraph G {
    ranksep=.25;
    edge [arrowsize=.5]
    ...
```

自然，此时只需将以“digraph G{”开头的字符串复制粘贴到graphviz中，就可以预览函数调用图。

在本节中，只用了少量代码我们就编写了一个函数调用图生成器。为了展示Cymbol语法的复用性，在下一节中，我们将在不修改它的情况下利用它来编写一个完全不同的程序。除此之外，我们还会用两个不同的监听器对同一棵语法分析树进行两趟遍历。

8.4 验证程序中符号的使用

在为类似Cymbol的编程语言编写解释器、编译器或者翻译器之前，我们需要确保Cymbol程序中使用的符号（标识符）用法正确。在本节中，我们计划编写一个能做出以下校验的Cymbol验证器：

- 引用的变量必须有可见的（在作用域中）定义
- 引用的函数必须有定义（函数可以以任何顺序出现，即函数定义提升）
- 变量不可用作函数
- 函数不可用作变量

要满足以上全部条件，我们需要做一点工作，因此理解本例可能会花费比其他例子更多的时间。不过，我们的收获将为编写真实的语言处理工具奠定坚实基础。

让我们首先来看一些包含不同标识符引用的样例代码，其中一些标识符是无效的。

```
listeners/vars.symbol
```

```
int f(int x, float y) {
    g();    // 前向引用是没问题的
    i = 3;  // 错误: i 未定义
    g = 4;  // 错误: g 不是变量
    return x + y; // x, y 已定义, 因此是正确的
}

void g() {
    int x = 0;
    float y;
    y = 9; // y 已定义
    f();   // 后向引用是没问题的
    z();   // 错误: 无此函数
    y();   // 错误: y 不是函数
    x = f; // 错误: f 不是变量
}
```

为验证一个程序中的所有内容都符合先前的定义，我们需要打印函数的列表和它们的局部变量，再加上全局符号（函数和全局变量）。此外，我们应该在发现问题的时候给出一个错误。例如，对于之前的输入，让我们编写一个名为**CheckSymbols**的程序，它将产生下列输出：

```
⇒ $ java CheckSymbols vars.symbol
< locals:[]
function<f:tINT>:[<x:tINT>, <y:tFLOAT>]
locals:[x, y]
function<g:tVOID>:[]
globals:[f, g]
line 3:4 no such variable: i
line 4:4 g is not a variable
line 13:4 no such function: z
line 14:4 y is not a function
line 15:8 f is not a variable
```

解决该问题的关键在于一种恰当的数据结构，称为符号表。我们的程序会将符号存储在符号表里，然后通过它来检查标识符引用的正确

性。在下一节中，我们将会看到这种数据结构，并利用它来解决验证符号的难题。

1.符号表速成

语言的实现者通常把存储符号的数据结构称为符号表。实现这样的语言意味着建立复杂的符号表结构。如果一门语言允许相同的标识符在不同的上下文中具备不同含义，那么对应的符号表实现就需要将符号按照作用域分组。一个作用域仅仅是一组符号的集合，例如一组函数的参数列表或者全局作用域中定义的变量和函数。

符号表本身仅仅是符号定义的仓库——它不进行任何验证工作。我们需要按照之前确定的规则，检查表达式中引用的变量和函数，以完成代码的验证。符号验证的过程中有两种基本的操作：定义符号和解析符号。定义一个符号意味着将它添加到作用域中。解析一个符号意味着确定该符号引用了哪个定义。在某种意义上，解析一个符号意味着寻找“最接近”的符号定义。最接近的定义域就是最内层的代码块。例如，下面的Cymbol示例代码包含了不同作用域（以黑圈数字标记）下的符号定义。

listeners/vars2.cymbol

```
❶ int x;  
   int y;  
❷ void a()  
❸ {  
    int x;  
    x = 1; // x 解析到了当前作用域的 x，而非全局作用域的 x  
    y = 2; // y 在当前作用域中不存在，但是在全局作用域中解析成功  
❹    { int y = x; }  
  }  
❺ void b(int z)  
❻ { }
```

全局作用域❶包含了变量x和y，以及函数a（）和b（）。函数定义在全局作用域中，但是建立了新的作用域，该作用域包含函数的参数（如果有的话），参见❷和❺。函数内部作用域（❸和❻）也可以嵌套产生一个新的作用域。局部变量声明于嵌套在对应函数作用域中的局部作用域（❸、❹和❻）中。

由于符号x被定义了两次，我们无法避免在同一个集合中处理所有标识符时的冲突问题。这就是作用域存在的意义。我们维护一组作用域，在同一个作用域中一个标识符只允许被定义一次。我们还为每个作用域维护一个指向父作用域的指针，这样，我们就能在外层作用域中寻找符号定义。全部的作用域构成一棵树，如图8-2所示。

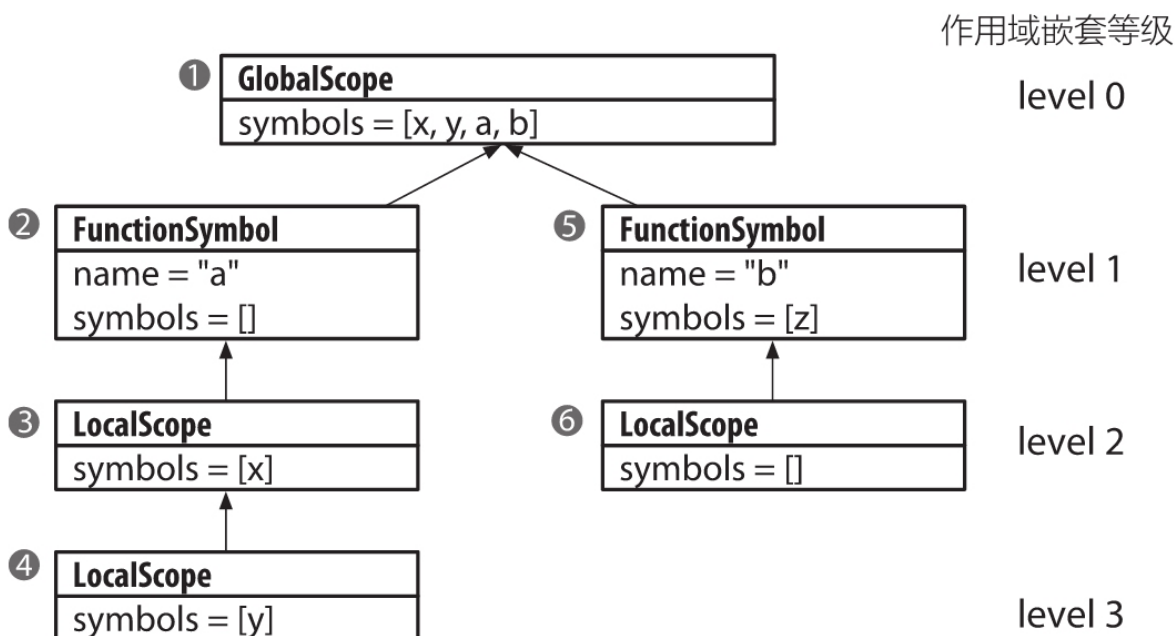


图8-2 全部作用域构成的树

圆圈中的数字代表源代码中的作用域。任何节点到根节点（全局作用域）的路径构成了一个作用域栈。当寻找一个符号定义时，我们从引用所在的作用域开始，沿着作用域树向上查找，直至找到其定义为止。

在本例中，为避免实现一个符号表的烦琐工作，我从参考文献

【Language Implementation Patterns[Par09]】一书的第6章中拷贝了符号表的源代码。我建议你阅读一下这份代码中的BaseScope、GlobalScope、LocalScope、Symbol、FunctionSymbol以及VariableSymbol，以熟悉符号表的实现过程。将这些类放在一起，就是

一个符号表了，我们暂且认为它们能够正常工作。有了符号表之后，我们就可以着手编写我们的验证器了。

2.验证器的架构

为完成该验证器，让我们从全局的角度进行一下规划。我们可以将这个问题分解为两个关键的操作：定义和解析。对于定义，我们需要监听变量和函数定义的事件，生成Symbol对象并将其加入该定义所在的作用域中。在函数定义开始时，我们需要将一个新的作用域“入栈”，然后在它结束时将该作用域“出栈”。

对于解析和校验符号引用，我们需要监听表达式中的变量和函数引用的事件。对于每个引用，我们要验证是否存在一个匹配的符号定义，以及该引用是否正确使用了该符号。虽然这种策略看上去相当直白，但是实际上存在一个难题：一个Cymbol程序可以在函数声明之前就调用它。我们称之为前向引用（forward reference）。为了支持这种情况，我们需要对语法分析树进行两趟遍历，第一趟遍历——或者说第一个阶段——对包括函数在内的符号进行定义，第二趟遍历中就可以看到文件中全部的函数了。下列代码触发了对语法分析树的两趟遍历：

```
listeners/CheckSymbols.java
```

```
ParseTreeWalker walker = new ParseTreeWalker();  
DefPhase def = new DefPhase();  
walker.walk(def, tree);  
// 新建一个阶段，将 def 中的符号表信息传递给该阶段  
RefPhase ref = new RefPhase(def.globals, def.scopes);  
walker.walk(ref, tree);
```

在定义阶段，我们将会创建很多个作用域。我们必须保持对这些定义域的引用，否则垃圾回收器会将它们清除掉。为保证符号表在从定义阶段到解析阶段的转换过程中始终存在，我们需要追踪这些作用域。最合乎逻辑的存储位置是语法分析树本身（或者使用一个将节点和值映射起来的标注Map）。这样，在沿语法分析树下降的过程中，查找一个引用对应的作用域就变得十分容易，因为函数或者局部代码块对应的树节点可以获得指向自身作用域的指针。

3.定义和解析符号

确定了全局的策略，我们就可以开始编写验证器了，不妨从DefPhase开始。它需要三个字段：一个全局作用域的引用、一个用于追踪我们创建的作用域的语法分析树标注器，以及一个指向当前作用域的指针。监听器方法enterFile（）启动了整个验证过程，并创建了一个全局作用域。最后的exitFile（）方法负责打印结果。

listeners/DefPhase.java

```
public class DefPhase extends CymbolBaseListener {
    ParseTreeProperty<Scope> scopes = new ParseTreeProperty<Scope>();
    GlobalScope globals;
    Scope currentScope; // 当前符号的作用域
    public void enterFile(CymbolParser.FileContext ctx) {
        globals = new GlobalScope(null);
        currentScope = globals;
    }

    public void exitFile(CymbolParser.FileContext ctx) {
        System.out.println(globals);
    }
}
```

当语法分析器发现一个函数定义时，我们的程序就需要创建一个 **FunctionSymbol** 对象。**FunctionSymbol** 对象有两项职责：作为一个符号，以及作为一个包含参数的作用域。为构造一个嵌套在全局作用域中的函数作用域，我们将一个函数作用域“入栈”。“入栈”是通过将当前作用域设置为该函数作用域的父作用域，并将它本身设置为当前作用域来完成的。

listeners/DefPhase.java

```
public void enterFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
    String name = ctx.ID().getText();
    int typeTokenType = ctx.type().start.getType();
    Symbol.Type type = CheckSymbols.getType(typeTokenType);

    // 新建一个指向外围作用域的作用域，这样就完成了入栈操作
    FunctionSymbol function = new FunctionSymbol(name, type, currentScope);
    currentScope.define(function); // 在当前作用域中定义函数
    saveScope(ctx, function);      // 入栈：设置函数作用域的父作用域为当前作用域
    currentScope = function;       // 现在当前作用域就是函数作用域了
}

void saveScope(ParserRuleContext ctx, Scope s) { scopes.put(ctx, s); }
```

方法 `saveScope()` 使用新建的函数作用域标注了该 `functionDecl` 规则节点，这样之后进行的下一个阶段就能轻易地获取相应的作用域。在函

数结束时，我们将函数作用域“出栈”，这样当前作用域就恢复为全局作用域。

```
listeners/DefPhase.java
public void exitFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
    System.out.println(currentScope);
    currentScope = currentScope.getEnclosingScope(); // 作用域“出栈”
}
```

局部作用域的实现与之类似。我们在监听器方法`enterBlock()` 中将一个作用域入栈，然后在`exitBlock()` 中将其出栈。

现在，我们已经能够很好地处理作用域和函数定义了，接下来让我们完成对参数和变量的定义。

```
listeners/DefPhase.java
public void exitFormalParameter(CymbolParser.FormalParameterContext ctx) {
    defineVar(ctx.type(), ctx.ID().getSymbol());
}

public void exitVarDecl(CymbolParser.VarDeclContext ctx) {
    defineVar(ctx.type(), ctx.ID().getSymbol());
}

void defineVar(CymbolParser.TypeContext typeCtx, Token nameToken) {
    int typeTokenType = typeCtx.start.getType();
    Symbol.Type type = CheckSymbols.getType(typeTokenType);
    VariableSymbol var = new VariableSymbol(nameToken.getText(), type);
    currentScope.define(var); // 在当前作用域中定义符号
}
```

这样，我们就完成了定义阶段代码的编写。

下面编写解析阶段的代码，首先，我们将当前作用域设置为定义阶段中得到的全局作用域。

```
listeners/RefPhase.java
```

```
public RefPhase(GlobalScope globals, ParseTreeProperty<Scope> scopes) {  
    this.scopes = scopes;  
    this.globals = globals;  
}  
public void enterFile(CymbolParser.FileContext ctx) {  
    currentScope = globals;  
}
```

之后，当树遍历器触发Cymbol函数和代码块的进入和退出方法时，我们根据定义阶段在树中存储的值，将currentScope设为对应的作用域。

```
listeners/RefPhase.java
```

```
public void enterFunctionDecl(CymbolParser.FunctionDeclContext ctx) {  
    currentScope = scopes.get(ctx);  
}  
public void exitFunctionDecl(CymbolParser.FunctionDeclContext ctx) {  
    currentScope = currentScope.getEnclosingScope();  
}  
  
public void enterBlock(CymbolParser.BlockContext ctx) {  
    currentScope = scopes.get(ctx);  
}  
public void exitBlock(CymbolParser.BlockContext ctx) {  
    currentScope = currentScope.getEnclosingScope();  
}
```

在遍历器正确设置作用域之后，我们就可以在变量引用和函数调用的监听器方法中解析符号了。当遍历器遇到一个变量引用时，它调用exitVar（），该方法使用resolve（）方法在当前作用域的符号表中查找该变量名。如果resolve方法在当前作用域中没有找到相应的符号，它会沿着外围作用域链查找。必要情况下，resolve将会一直向上查找，直至全局作用域为止。如果它没有找到合适的定义，则返回null。此外，若resolve（）方法找到的符号是函数而非变量，我们就需要生成一个错误消息。

```
listeners/RefPhase.java
```

```
public void exitVar(CymbolParser.VarContext ctx) {
    String name = ctx.ID().getSymbol().getText();
    Symbol var = currentScope.resolve(name);
    if ( var==null ) {
        CheckSymbols.error(ctx.ID().getSymbol(), "no such variable: "+name);
    }
    if ( var instanceof FunctionSymbol ) {
        CheckSymbols.error(ctx.ID().getSymbol(), name+" is not a variable");
    }
}
```

处理函数调用的方法与之基本相同。如果找不到定义，或者找到的定义是变量，那么我们就输出一个错误。

最后，下面的构建和测试过程能够产生之前预期的输出结果。

```
$ antlr4 Cymbol.g4
$ javac Cymbol*.java CheckSymbols.java *Phase.java *Scope.java *Symbol.java
$ java CheckSymbols vars.cymbol
locals:[]
function<f:tINT>:[<x:tINT>, <y:tFLOAT>]
...
```

在完成两趟遍历的代码后，我们的验证器就大功告成了。此过程涉及了很多处理细节，不过这些努力都是值得的，因为它是编写你自己的语言处理工具的一个很好的起点。本例中监听器的实现只有区区150行Java代码，符号表的实现也仅有100行。如果你还不急于编写一个需要符号表的程序，那就无须在意它的细节。重点在于，一种用于追踪和验证符号的广为人知的符号表实现已经存在了，它并不是什么高深莫测的技术。欲了解更多符号表管理的内容，我不客气地推荐你购买和阅读我的【Language Implementation Patterns[Par09]】一书。

如果你在迄今为止的阅读过程中都能保持很好的同步，那么你的状态不错！现在，你不仅可以根据语言的参考手册构建语法，还可以赋予这些语法生命——通过实现监听器来完成有用的工作。当然，你现在可能遇到了一些难题，不过你的功力已经相当深厚了。

本章是本书第二部分的最后一章。当你掌握了关键的ANTLR技能之后，你会期待下一部分带来的ANTLR高级用法的。

本书由“ePUBw.COM”整理，ePUBw.COM 提供
最新最全的优质电子书下载！！！！

第三部分 高级特性

在第二部分中，我们学习了如何从范例代码和参考手册中提取一门语言的抽象结构（句法），并使用ANTLR对该句法进行正式表述。为了开发语言类应用程序，我们编写了一些树监听器和访问器，用它们操纵自动生成的语法分析树。这样，我们就掌握了使用ANTLR来高效地处理大多数问题的关键技巧。

第三部分的主要内容是ANTLR的高级用法。首先，我们将会学习ANTLR的自动异常处理机制。其次，我们将会探究如何在语法中直接嵌入代码片段，以便在解析过程中实时地产生输出或者执行计算。再

次，我们将会看到如何基于运行时信息，通过语义判定来动态开启或者关闭语法中的备选分支。最后，我们将会介绍一些词法方面的“黑魔法”。

第9章 错误报告与恢复

同绝大多数软件一样，在我们开发一门语法的过程中，需要修复很多的错误。直到我们编写完（并调试完）语法之后，生成的语法分析器才能识别所有的有效输入语句。在这个过程中，**ANTLR**的错误消息含有丰富的信息，有助于我们调试语法中产生的问题。一旦拥有了正确的语法，我们就必须处理不合语法的语句，这些语句可能来源于用户输入，甚至是其他程序在错误情况下自动生成的。

在上述情况下，我们的语法分析器对非法输入的响应就会大大影响生产力。换句话说，无论在我们的开发过程中，还是在用户的实际使用过程中，一个只会输出“呃，出错了”并且一遇到语法错误就退出的语法分析器毫无用处。

使用**ANTLR**的开发者将会无偿获得它提供的优秀的错误报告功能和复杂的错误恢复机制。**ANTLR**生成的语法分析器能够自动地在遇到句法错误时产生丰富的错误消息，并且能在大多数情况下成功地完成重新同步。这样的语法分析器甚至能够保证只为每个句法错误产生一条错误消息。在本章中，我们将会学习**ANTLR**自动生成的语法分析器使用

的自动错误报告和恢复策略。我们还将了解如何修改默认的错误处理机制，使之符合一些典型情况下的需求，以及如何在特定的程序中定制错误消息。

9.1 错误处理入门

描述ANTLR的错误恢复策略，最好的方法是观察一个ANTLR自动生成的语法分析器对错误输入产生的响应。下面让我们看一个简单的类Java语言的语法，它的类定义中包含字段和方法，其中的方法具有简单的语句和表达式。该语法将成为本节和本章中其余各节的例子的核心。

```
errors/Simple.g4
```

```
grammar Simple;
```

```
prog:   classDef+ ; // 匹配一个或多个类定义
```

```
classDef
```

```
    :   'class' ID '{' member+ '}' // 一个类具有若干个成员
        {System.out.println("class " + $ID.text);}
    ;
```

```
member
```

```
    :   'int' ID ';' // 字段定义
        {System.out.println("var " + $ID.text);}
    |   'int' f=ID '(' ID ')' '{' stat '}' // 方法定义
        {System.out.println("method: " + $f.text);}
    ;
```

```
stat:
```

```
    expr ';'
        {System.out.println("found expr: " + $stat.text);}
    |   ID '=' expr ';'
        {System.out.println("found assign: " + $stat.text);}
    ;
```

```
expr:
```

```
    INT
    |   ID '(' INT ')'
    ;
```

```
INT :   [0-9]+ ;
```

```
ID  :   [a-zA-Z]+ ;
```

```
WS  :   [ \t\r\n]+ -> skip ;
```

(注: ANTLR 4.3之后, 原先的\$stat.text需要改成\$ctx.getText ()。

——译者注)

其中的内嵌动作会打印出语法分析器发现的相应元素。出于方便和简洁的目的, 我们使用内嵌动作来代替语法分析树监听器。我们将会在第10章中学习更多有关动作的知识。

首先, 我们运行语法分析器, 给出一些正确的输入, 观察正常情况下的输出。

```
⇒ $ antlr4 Simple.g4
⇒ $ javac Simple*.java
⇒ $ grun Simple prog
⇒ class T { int i; }
⇒ EOF
< var i
  class T
```

我们没有从语法分析器中得到任何错误，它正常地执行了任务并使用打印结果报告，对变量*i*和类定义*T*的识别已经成功完成。

现在，让我们试着输入一个类，它的方法定义中包含一个非法的赋值语句。

```
⇒ $ grun Simple prog -gui
⇒ class T {
⇒   int f(x) { a = 3 4 5; }
⇒ }
⇒ EOF
< line 2:19 mismatched input '4' expecting ';'
  method: f
  class T
```

在词法符号4处，语法分析器没有发现期望的“;”，因此报告了一个错误。输出的line 2: 19表明，有误的词法符号位于第2行的第20个字符处（字符位置从0开始）。因为“-gui”选项的存在，我们还可以看到一棵将错误节点高亮显示（稍后会讲到）的语法分析树，如图9-1所示。

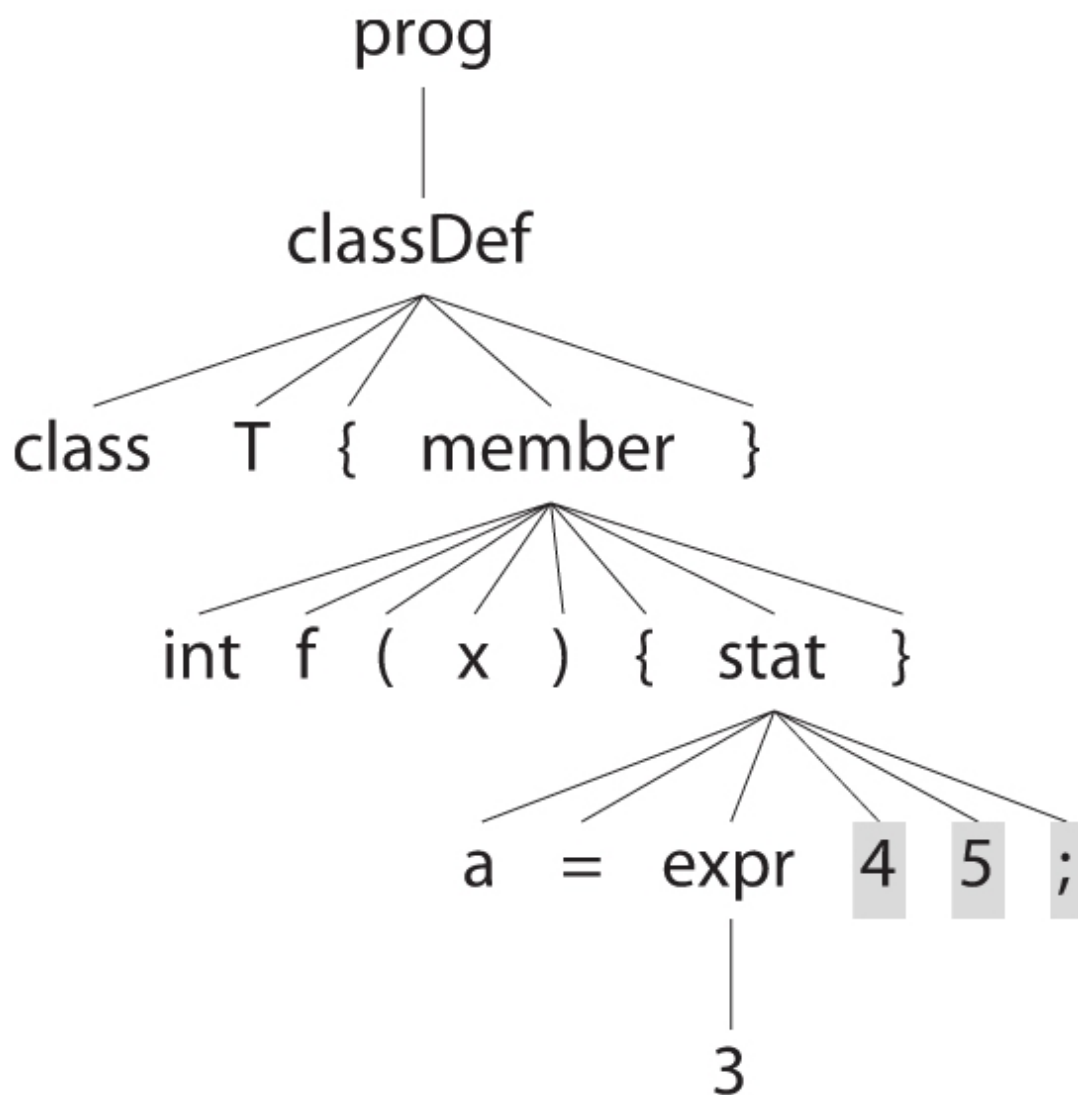


图9-1 错误节点高亮显示的语法分析树

在这个例子中，输入包含两个多余的词法符号，因此，语法分析器针对这样的错误给出了一个通用的错误信息。不过，如果输入仅有一个多余的词法符号，语法分析器就能够表现得更加智能，指出存在一个多余的词法符号。在下面的测试中，在类名和类定义体之间存在一个多余的“;”：

```

⇒ $ grun Simple prog
⇒ class T ; { int i; }
⇒ EOF
< line 1:8 extraneous input ';' expecting '{'
    var i
    class T

```

语法分析器在“;”处报告了一个错误，并且给出了一个信息量更大的结果，因为它知道“;”后面的词法符号是自己期望看到的。这个特性叫作单词法符号移除（single-token deletion），实现这个特性只需要语法分析器假设多余的那个词法符号不存在，然后继续解析过程即可。

同样，在语法分析器检测到词法符号缺失的时候，它也可以完成单词法符号补全（single-token insertion）。下面让我们去掉最后的“}”，看看会发生什么。

```

⇒ $ grun Simple prog
⇒ class T {
⇒   int f(x) { a = 3; }
⇒ EOF
< found assign: a=3;
    method: f
    line 3:0 missing '}' at '<EOF>'
    class T

```

与编程语言理论有关的幽默二则

显然，伟大的计算机科学家Niklaus Wirth极富幽默感。他曾经开玩笑说，欧洲人以“传引用”方式称呼他（欧洲人通常能将他的名字正确读

作“Ni-klaus Virt”），美国人以“传值”方式称呼他（将他的名字误读作“Nickle-less Worth”）。

在Compiler Construction 1994会议上，Kristen Nygaard（Simula的发明者）讲了一个故事，有一次在一门编程语言的理论课上，他说，“强类型（strong typing）是法西斯主义”，意指自己偏好弱类型的编程语言。后来，一个学生问他，为什么用力打字（strong typing）是法西斯主义。

语法分析器报告它未能找到结尾的“}”词法符号。

另外一种常见的句法错误发生在语法分析器做出决策的关键位置，剩余的输入文本不符合规则的任意一个备选分支。例如，如果我们在字段声明中遗漏了变量名，`member`规则的两个备选分支就都无法匹配这样的输入。因此，语法分析器报告没有找到可行的备选分支。

```
⇒ $ grun Simple prog
⇒ class T { int ; }
⇒ E0
  < line 1:14 no viable alternative at input 'int;'
    class T
```

错误报告中的“`int`”和“`;`”之间没有空格，这是因为，我们令词法分析器在空白符号的`WS ()`规则中执行了`skip ()`指令。

如果存在词法错误，ANTLR也会给出一个错误消息，指明它无法将一个或者多个字符匹配为词法符号。例如，如果输入一个完全未知的字

符，我们就会得到一个词法符号识别错误。

```
⇒ $ grun Simple prog
⇒ class # { int i; }
⇒ E0
  < line 1:6 token recognition error at: '#'
    line 1:8 missing ID at '{'
    var i
    class <missing ID>
```

由于我们没有给出一个有效的类名，单词法符号补全机制生成了一个 **missing ID** 作为类名，这样，类名的词法符号就不至于为空了。如果需要控制语法分析器对这样的词法符号的生成机制，覆盖

DefaultErrorStrategy 类中的 **getMissingSymbol** () 方法即可（参见9.5节）。

你可能注意到了，本节中的示例代码显示，尽管发生了错误，语法分析过程还是照常进行。除了产生良好的错误消息和利用剩余的输入进行重新同步之外，语法分析器还必须能够移动到合适的位置继续语法分析过程。

例如，当通过 **classDef** 规则中的子规则 **member** 匹配类成员时，语法分析器不应该在遇到非法的成员定义时结束 **classDef**。这就是语法分析器能够跳过错误的原因——一个句法错误不应该让语法分析器结束当前规则。语法分析器将会尽最大可能匹配到一个合法的类定义。我们将会

在9.3节深入研究这个主题。不过，首先让我们来看看如何修改标准的错误报告机制，以利于调试语法和为用户提供更恰当的消息。

9.2 修改和转发ANTLR的错误消息

默认情况下，ANTLR将所有的错误消息送至标准错误（`standard error`），不过我们可以通过实现接口`ANTLRErrorListener`来改变这些消息的目标输出和内容。该接口有一个同时应用于词法分析器和语法分析器的`syntaxError()`方法。`syntaxError()`方法接收各式各样的信息，无论是错误的位置还是错误的内容。它还接收指向语法分析器的引用，因此我们能够通过该引用来查询识别过程的状态。

例如，下列错误监听器（`error listener`）来自于文件 `TestE_Listener.java`，能够在通常的带有词法符号信息的错误消息后面打印出规则的调用栈：

```
errors/TestE_Listener.java
```

```
public static class VerboseListener extends BaseErrorListener {
    @Override
    public void syntaxError(Recognizer, ? recognizer,
                           Object offendingSymbol,
                           int line, int charPositionInLine,
                           String msg,
                           RecognitionException e)
    {
        List<String> stack = ((Parser)recognizer).getRuleInvocationStack();
        Collections.reverse(stack);
        System.err.println("rule stack: "+stack);
        System.err.println("line "+line+"."+charPositionInLine+" at "+
                           offendingSymbol+": "+msg);
    }
}
```

使用这种方法，我们的程序就能在语法分析器调用起始规则之前，轻易地为其增加一个错误监听器。

```
errors/TestE_Listener.java
```

```
SimpleParser parser = new SimpleParser(tokens);
parser.removeErrorListeners(); // 移除 ConsoleErrorListener
parser.addErrorListener(new VerboseListener()); // 增加我们自定义的错误监听器
parser.prog(); // 照常进行解析过程
```

在我们增加自定义的错误监听器之前，我们需要移除输出目标是控制台的内置错误监听器，以防出现重复的错误消息。

让我们输入一个特殊的、包含多余的类名且缺失字段名的类定义，看看现在的错误消息。

```

⇒ $ javac TestE_Listener.java
⇒ $ java TestE_Listener
⇒ class T T {
⇒   int ;
⇒ }
⇒ EOF
  < rule stack: [prog, classDef]
    line 1:8 at [@2,8:8='T',<9>,1:8]: extraneous input 'T' expecting '{'
    rule stack: [prog, classDef, member]
    line 2:6 at [@5,18:18=';',<8>,2:6]: no viable alternative at input 'int;'
    class T

```

其中，栈内容[prog, classDef]显示，语法分析器当前正处于规则classDef中，该规则是由prog调用的。注意词法符号的信息还包括对应的词法符号在输入字符流中的位置。这有助于在类似开发环境之类的输入中对错误进行高亮显示。例如，词法符号[@2, 8: 8='T', <9>, 1: 8]显示，它是词法符号流中的第三个（索引是2，从0开始计数），包含的字符索引由8到8，词法符号类型为9，位于第1行第8个字符处（从0开始计数，tab符看作一个字符）。

通过Java Swing技术，我们可以非常容易地将这条消息使用一个对话框来显示，只需要修改一下syntaxError () 方法即可。

errors/TestE_Dialog.java

```
public static class DialogListener extends BaseErrorListener {
    @Override
    public void syntaxError(Recognizer, ? recognizer,
                           Object offendingSymbol,
                           int line, int charPositionInLine,
                           String msg,
                           RecognitionException e)
    {
        List<String> stack = ((Parser)recognizer).getRuleInvocationStack();
        Collections.reverse(stack);
        StringBuilder buf = new StringBuilder();
        buf.append("rule stack: "+stack+" ");
        buf.append("line "+line+": "+charPositionInLine+" at "+
                   offendingSymbol+": "+msg);

        JDialog dialog = new JDialog();
        Container contentPane = dialog.getContentPane();
        contentPane.add(new JLabel(buf.toString()));
        contentPane.setBackground(Color.white);
        dialog.setTitle("Syntax error");
        dialog.pack();
        dialog.setLocationRelativeTo(null);
        dialog.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        dialog.setVisible(true);
    }
}
```

使用输入class T{int int i; }来测试TestE_Dialog，可以看到如图9-2所示的对话框。

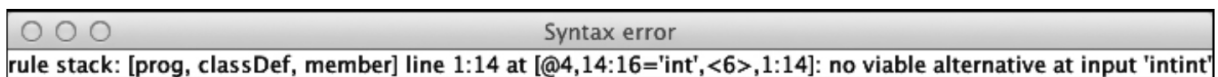


图9-2 测试TestE_Dialog对话框

请看下一个例子，构建一个错误监听器TestE_Listener2.java，用下划线标示出有问题的词法符号，如下所示：


```

⇒ $ javac TestE_Listener2.java
⇒ $ java TestE_Listener2
⇒ class T XYZ {
⇒   int ;
⇒ }
⇒ EOF
< line 1:8 extraneous input 'XYZ' expecting '{'
  class T XYZ {
        ^^^
line 2:6 no viable alternative at input 'int;'
  int ;
    ^
class T

```

为简单起见，我们将忽略tab符——`charPositionInLine`并不是实际的列数，因为tab符并没有统一的宽度。下面的错误监听器实现用下划线标示出了错误的位置，正如之前我们所看到的那样：

errors/TestE_Listener2.java

```

public static class UnderlineListener extends BaseErrorListener {
    public void syntaxError(Recognizer<?, ?> recognizer,
                           Object offendingSymbol,
                           int line, int charPositionInLine,
                           String msg,
                           RecognitionException e)

    {
        System.err.println("line "+line+": "+charPositionInLine+" "+msg);
        underlineError(recognizer, (Token)offendingSymbol,
                       line, charPositionInLine);
    }

    protected void underlineError(Recognizer recognizer,
                                  Token offendingToken, int line,
                                  int charPositionInLine) {
        CommonTokenStream tokens =
            (CommonTokenStream)recognizer.getInputStream();
        String input = tokens.getTokenSource().getInputStream().toString();
    }
}

```

```

        String[] lines = input.split("\n");
        String errorLine = lines[line - 1];
        System.err.println(errorLine);
        for (int i=0; i<charPositionInLine; i++) System.err.print(" ");
        int start = offendingToken.getStartIndex();
        int stop = offendingToken.getStopIndex();
        if ( start>=0 && stop>=0 ) {
            for (int i=start; i<=stop; i++) System.err.print("^");
        }
        System.err.println();
    }
}

```

我们还需要了解有关错误监听器的最后一件事情。当语法分析器检测到有歧义的输入序列时，它会通知错误监听器。默认的错误监听器 `ConsoleErrorListener` 不会向控制台打印任何东西。正如我们在2.3节中所看到的那样，有歧义的输入可能意味着我们的语法存在错误，语法分析器不应该因此通知用户。下面让我们来回顾一下该节中有歧义的语法匹配“f () ; ”的两种不同方式。

```

errors/Ambig.g4
grammar Ambig;

stat: expr ';'          // 表达式语句
    | ID '(' ')' ';'    // 函数调用语句
    ;

expr: ID '(' ')'
    | INT
    ;

INT :  [0-9]+ ;
ID  :  [a-zA-Z]+ ;
WS  :  [ \t\r\n]+ -> skip ;

```

如果我们用这个语法进行测试，我们不会看到有关歧义的公告。

```
⇒ $ antlr4 Ambig.g4
⇒ $ javac Ambig*.java
⇒ $ grun Ambig stat
⇒ f();
⇒ EOf
```

当语法分析器检测到歧义发生时，如果希望得到通知，请使用 `addErrorListener ()` 方法添加一个 `DiagnosticErrorListener` 的实例来告知语法分析器。

```
parser.removeErrorListeners(); // 移除 ConsoleErrorListener
parser.addErrorListener(new DiagnosticErrorListener());
```

此外，你还应当告诉语法分析器，你对所有的歧义警告都感兴趣，而不仅仅是那些可以快速检测到的。出于效率方面的原因，**ANTLR** 的决策机制并不是总能发现所有的歧义信息。下面是令语法分析器报告所有歧义的方法：

```
parser.getInterpreter()
    .setPredictionMode(PredictionMode.LL_EXACT_AMBIG_DETECTION);
```

如果你在用 `grun` 命令运行 `TestRig`，加上选项“`-diagnostics`”令其使用 `DiagnosticError-Listener` 替代默认的控制台错误监听器（并打开 `LL_EXACT_AMBIG_DETECTION` 选项）即可。

```
⇒ $ grun Ambig stat -diagnostics
⇒ f();
⇒ EOf
< line 1:3 reportAttemptingFullContext d=0, input='f()';'
  line 1:3 reportAmbiguity d=0: ambigAlts={1, 2}, input='f()';'
```

输出结果显示语法分析器还调用了`reportAttemptingFullContext()`。

ANTLR在SLL(*)分析失败时调用此方法，语法分析器会启用功能更加强大的完整ALL(*)机制。详见13.7节。

在开发过程中使用上面提到的诊断错误监听器（`diagnostics error listener`）是个好主意，因为ANTLR工具（在生成语法分析器时）不会对歧义性语法结构提出静态警告。在ANTLR 4中，只有运行状态的语法分析器才能检测到歧义。这就像是Java中静态类型机制和Python中动态类型机制的差别。

ANTLR 4的若干项改进

在ANTLR 4中，有两个与错误处理相关的重大改进：ANTLR的内置错误恢复机制更加优秀，同时也让开发者能够更加容易地修改错误处理策略。当Sun公司使用ANTLR 3编写JavaFX的语法分析器时，他们注意到一个放错位置的分号会导致语法分析器在搜寻一系列元素（例如通过`member+`定义的类成员）的过程中提前终止。现在，ANTLR 4的语法分析器会试图在子规则的认识之前和认识过程中进行重新同步

（`resynchronize`），而非草草丢弃词法符号并退出当前规则。第二项改进允许开发者按照策略模式（`Strategy pattern`）指定自定义的错误处理机制。

现在，我们已经深入了解了ANTLR语法分析器产生的消息类型，以及修改和转发它们的方法，接下来，让我们探索一下错误恢复方面的知识。

9.3 自动错误恢复机制

错误恢复指的是允许语法分析器在发现语法错误后还能继续的机制。原则上，最好的错误恢复来自人类在手工编写的递归下降的语法分析器中进行的干预。尽管如此，按照我的经验，手工编写一个优秀的错误恢复机制非常难，因为这个过程过于枯燥乏味，极易出错。在本书描述的ANTLR最新版中，我穷尽我毕生所学，基于多年的经验，来为ANTLR语法提供良好的错误恢复机制。

ANTLR的错误恢复机制基于Niklaus Wirth的早期著作【Algorithms + Data Structures = Programs[Wir78]】中的思想（以及Rodney Topor的【A Note on Error Recovery in Recursive Descent Parsers[Top82]】，同时也包含Josef Grosch在他的CoCo语法分析器生成器中的优秀思想【Efficient and Comfortable Error Recovery in Recursive Descent Parsers[Gro90]】。

下面是ANTLR将这些思想糅合在一起的实现细节：必要情况下，语法分析器在遇到无法匹配词法符号的错误时，执行单词法符号补全和单词法符号移除。如果这些方案不奏效，语法分析器将向后查找词法符号，直到它遇到一个符合当前规则的后续部分的合理词法符号为止，

接着，语法分析器将会继续语法分析过程，仿佛什么事情都没有发生过一样。在本节中，我们将会看到上述术语的含义，并探究ANTLR是如何在错综复杂的情况下从错误中恢复的。下面让我们首先分析ANTLR使用的基本错误恢复策略。

1.通过扫描后续词法符号来恢复

当面对真正的非法输入时，当前的规则无法继续下去，此时语法分析器将会向后查找词法符号，直到它认为自己已经完成重新同步时，它就返回原先被调用的规则。我们可以称为同步-返回（**sync-and-return**）策略。有人称为“应急模式”（**panic mode**），不过它的表现相当好。语法分析器知道自己无法使用当前规则匹配当前输入。它会持续丢弃后续词法符号，直至发现一个可以匹配本规则中断位置之后的某条子规则的词法符号。例如，如果在赋值语句中存在一个语法错误，那么语法分析器的做法就非常合理：丢弃后续的词法符号，直到发现一个分号或者其他的语句终结符为止。这种策略较为激进，但是十分有效。我们下面将要看到，这种基本策略作为后备方案，在启用之前，ANTLR会试图在规则内部进行恢复。

每个ANTLR自动生产的规则方法都被包裹在一个**try-catch**块内，它应对语法错误的措施是报告该错误，并试图在返回之前从该错误中恢复。

```
try {  
    ...  
}  
catch (RecognitionException re) {  
    _errHandler.reportError(this, re);  
    _errHandler.recover(this, re);  
}
```

我们将会在第9.5节中看到错误处理策略的更多细节，不过，就现在而言，我们可以认为`recover()`会持续消费词法符号，直到发现重新同步集合（**resynchronization set**）中的词法符号为止。重新同步集合是调用栈中所有规则的后续符号集合（**following set**）的并集。一条规则引用（**rule reference**）的后续符号集合是能够立即延续该规则，从而无须离开当前规则的词法符号集合。例如，给定一个备选分支`assign'; '`，那么规则引用`assign`的后续符号集合就是`{'; '}`。如果该备选分支是`assign`，那么后续符号集合就是空的。

有必要通过一个例子来加深对重新同步集合的理解。请看下列语法，想象一下，在每条规则的调用过程中，语法分析器都会追踪每次规则调用的后续符号集合。

errors/F.g4

```
grammar F;  
group  
  : '[' expr ']'      // expr 规则引用的后续词法符号：{' '}'  
  | '(' expr ')'      // expr 规则引用的后续词法符号：{' '}'  
  ;  
expr: atom '^' INT ;   // atom 规则引用的后续词法符号：{' '^'}  
atom: ID  
      | INT  
      ;  
  
INT : [0-9]+ ;  
ID  : [a-zA-Z]+ ;  
WS  : [ \t\r\n]+ -> skip ;
```

请看输入文本[1^2]对应的如图9-3左侧所示的语法分析树：

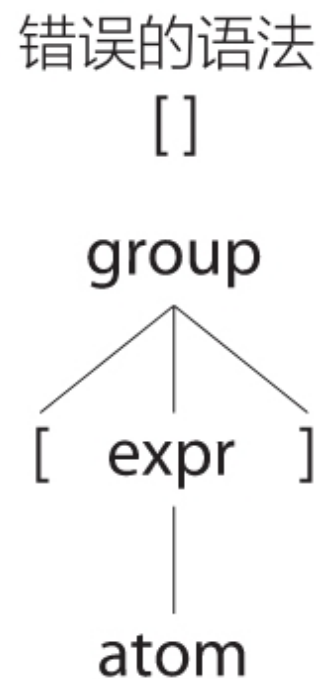
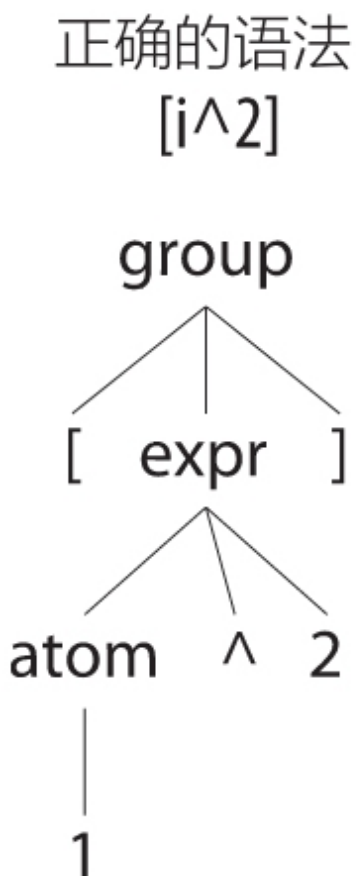


图9-3 某语法分析树

当匹配规则`atom`中的词法符号1时，调用栈是`[group, expr, atom]`（这是因为`group`调用了`expr`，后者又调用了`atom`）。通过查看调用栈，我们就能清楚地知道语法分析器抵达此处时，紧跟在每条被其调用的规则后面的词法符号的集合。后续符号集合只考虑那些在当前规则中出现的词法符号，因此，在运行时，我们可以只把当前调用栈对应的后续符号集合组合在一起。换句话说，我们无法同时途径`group`的两个备选分支来到规则`expr`处。

语法`F`中的注释里给出了一些后续符号集合，将它们组合在一起，我们得到了上述输入的重新同步集合`{'^', ']'}`。为了证明该集合是符合预期的，让我们看看当语法分析器遇到错误输入`[]`时会发生些什么。此时，我们会得到图9-3中右侧的语法分析树。在`atom`中，语法分析器发现当前词法符号`]`不符合`atom`的任意两个备选分支之一。为了完成重新同步，语法分析器将持续消费词法符号，直到它发现重新同步集合中的词法符号为止。在本例中，当前的词法符号`]`正好是重新同步集合的成员之一，因此语法分析器实际上没有消费任何词法符号就完成了在`atom`中的重新同步。

在完成`atom`规则中的恢复过程后，语法分析器返回`expr`规则，但是它立即发现缺少`^`词法符号。上述恢复过程将会重复，语法分析器将持续消费词法符号，直到发现`expr`规则的重新同步集合中的元素为止。`expr`规则的重新同步集合，也就是`group`规则的第一个备选分支中引用的`expr`的

后续符号集合，即{'}'。再一次，语法分析器没有消费任何东西就退出了`expr`规则，返回到了`group`规则的第一个备选分支中。现在语法分析器知道自己找到了`expr`规则引用之后的内容——它成功地匹配到了`group`规则中的'}'，这样，语法分析器就成功地完成了重新同步。

在恢复过程中，**ANTLR**语法分析器会避免输出层叠的错误消息（从**Grosch**中借鉴的思想）。即，对于每个语法错误，直到成功从该错误中恢复，语法分析器才输出一条错误消息。这件事情是通过一个简单的布尔类型的变量完成的，若该变量被置为`true`，当遇到语法错误时，语法分析器就能避免输出进一步的错误，直到语法分析器成功地匹配到一个词法符号，或者该变量被置为`false`为止（参见**DefaultErrorStrategy**类中的`errorRecoveryMode`字段）。

紧随其后的符号集合（**FOLLOW Set**）vs.后续符号集合（**Following Set**）

熟悉编程语言理论的读者可能会有疑问，`atom`规则的重新同步集合是否应该是紧跟在`atom`之后的词法符号（用**FOLLOW (atom)**表示）集合，即所有能在某种上下文中紧跟在`atom`之后的词法符号集合？非常不幸的是，事情没有那么简单，要想用在特定上下文而非全部上下文中可能跟随在某规则之后的词法符号集合构建重新同步集合，必须通过动态计算。**FOLLOW (expr)**是{'}', '}', '}'，它包含了在所有可能的上下文中（`group`的第一条和第二个备选分支）紧跟着`expr`规则引用的

词法符号。很显然，尽管如此，在运行时，语法分析器同时只能从一个位置调用`expr`。注意到`FOLLOW (atom)`是`'^'`，如果语法分析器使用这个词法符号而非重新同步集合`{' ', '\n'}`来进行重新同步，它可能会持续消费词法符号，直到文件的末尾，因为输入内容中并没有`^`。

在许多情况下，`ANTLR`能够更加智能地完成恢复，而不仅仅是本节中提到的“寻找重新同步集合中的符号”和“从当前规则返回”。它会尽力尝试“修复”输入文本并继续相同规则。在下面几个小节中，我们将会看到语法分析器是如何从错误匹配的词法符号和子规则的错误中恢复的。

2.从不匹配的词法符号中恢复

在语法分析的过程中，最常见的操作之一就是“匹配词法符号”。对于语法中的每个词法符号`T`，语法分析器都会调用`match (T)`。如果当前的词法符号不是`T`，`match ()`方法就会通知错误监听器，并试图重新同步。为完成同步，它有三种选择：移除一个词法符号、补全一个词法符号，或者简单地抛出一个异常以启用基本的同步-返回机制。

如果能够成功的话，移除当前的词法符号是重新同步最容易的方法。让我们回顾一下之前`Simple`语法定义的“简单类定义语言”里的`classDef`规则。

```
errors/Simple.g4
```

```
classDef
```

```
    : 'class' ID '{' member+ '}' // a class has one or more members
    {System.out.println("class "+$ID.text);}
    ;
```

考虑输入文本`class 9 T{int i; }`，语法分析器会删除`9`，然后继续进行同一条规则的语法分析过程——匹配类的定义体。图9-4展示了语法分析器在分析完`class`时的状态。

LA (1) 和LA (2) 标示出了第一个和第二个（在当前词法符号之后的）前瞻词法符号。`match (ID)` 期望LA (1) 是一个ID，但是它不是。不过，下一个词法符号LA (2) 是一个ID。此时，我们只需移除当前的词法符号（将它当作干扰项），然后按照预期匹配下一个ID并退出`match ()` 方法，即可完成恢复过程。

如果语法分析器无法通过移除一个词法符号的方式重新同步，它会转而尝试补全一个词法符号。假设我们忘记输入ID，那么`classDef`规则看到的输入就是`class{int i; }`。在匹配完`class`后，输出的状态如图9-5所示。

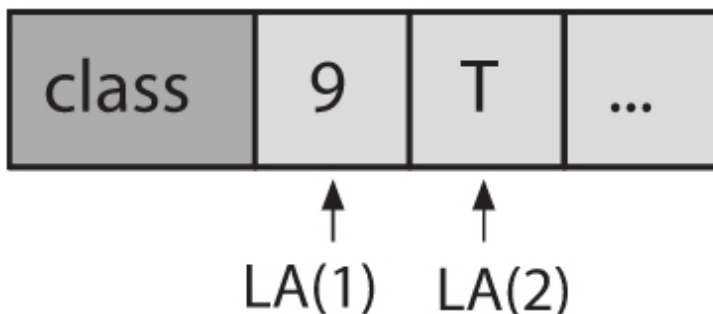


图9-4 语法分析器分析完class时的状态

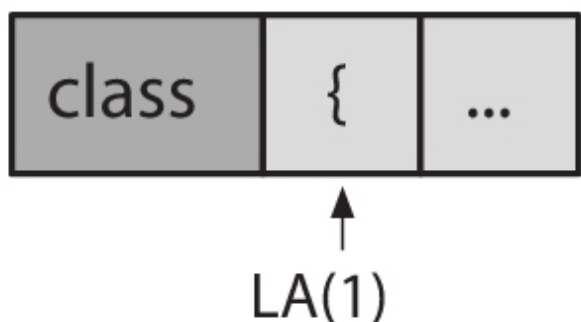


图9-5 匹配完class后的输出状态

语法分析器调用了`match (ID)`，期望发现一个标识符，但是实际上发现的却是`{`。在这种情况下，语法分析器知道`{`是自己所期望的那个词法符号的下一个，因为在`classDef`规则中它位于`ID`之后。此时`match ()`方法可以假定标识符已经被发现并返回，这样，下一个`match ('{')`的调用就会成功。

在忽略内嵌动作（例如打印出类名的语句）的前提下，这种方案表现得相当出色。但是，如果词法符号是`null`，通过`$ID.text`引用了缺失词法符号的打印语句就会引起一个异常。因此，错误处理器会创建一个词法符号，而非简单的假定该词法符号存在，详情参见

`DefaultErrorStrategy`中的`getMissingSymbol ()`方法。新创建的词法符号具有语法分析器所期望的类型，以及和当前词法符号`LA (1)`相同的

行列位置信息。这个新创建的词法符号阻止了监听器和访问器中引用缺失词法符号时引发的异常。

分析语法分析过程最容易的方法是查看语法分析树，它展示了语法分析器识别所有词法符号的细节。一旦遇到错误，语法分析树就会用红色高亮标注那些词法分析器在重新同步过程中移除或者补全的词法符号。对于输入文本`class{int i; }`和Simple语法，我们得到如图9-6所示的语法分析树。

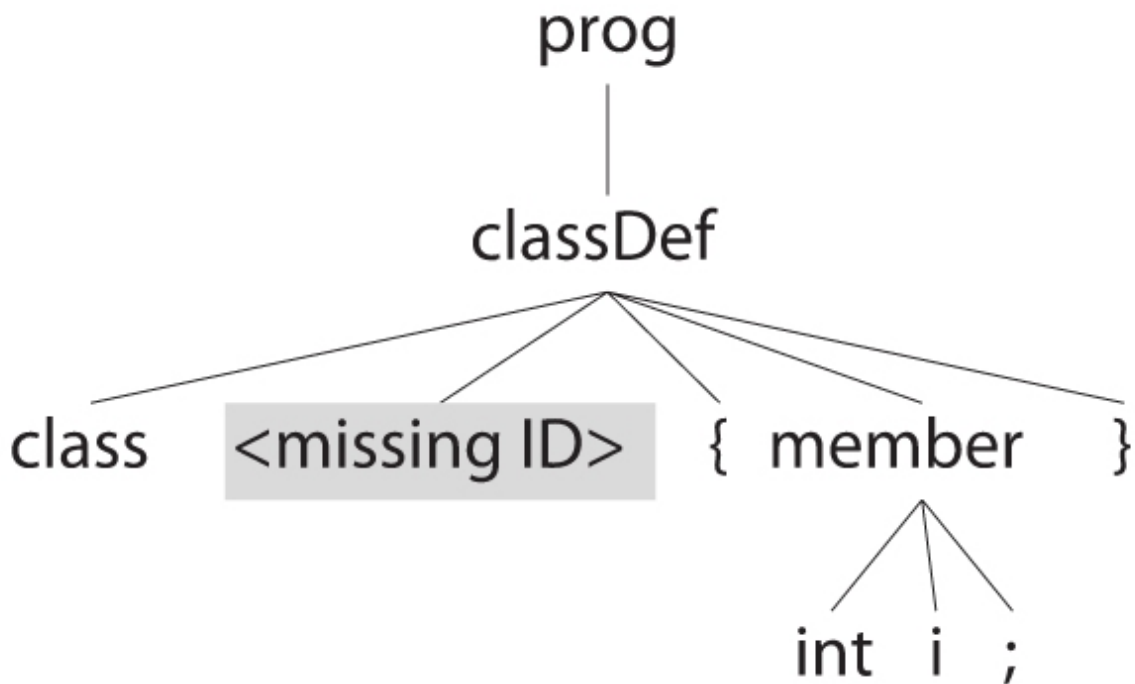


图9-6 输入文本`class{int i; }`和Simple语法后的语法分析树

同时，语法分析器执行了内嵌动作，成功地完成了打印而没有抛出异常，这是由于错误恢复机制为`$ID`创建了一个有效的Token对象。

```
⇒ $ grun Simple prog -gui
⇒ class { int i; }
⇒ E0F
  < line 1:6 missing ID at '{'
    var i
    class <missing ID>
```

显然，对我们的目的而言，一个<missing ID>标识符没有任何意义，不过，至少错误恢复机制不会引起一堆空指针异常了。

现在，我们已经知道了ANTLR针对简单的词法符号实施的规则内恢复机制，接下来，让我们进一步探索它在识别子规则之前以及子规则识别过程中的错误恢复机制。

3.从子规则的错误中恢复

许多年前，Sun公司的JavaFX小组向我反馈，他们使用的ANTLR自动生成的语法分析器在特定情况下无法很好地从错误中恢复。实际情况是，语法分析器在遇到第一个错误时就退出了类似member+的子规则循环，从而强制将同步-返回机制作用于外围规则。例如，“var width Number;”（width后面缺少冒号）这样一个有关成员声明的小错误就会令语法分析器忽略后续的全部成员。

Jim Idle是一个ANTLR邮件组内的贡献者和顾问，他提出了一种我称为“Jim Idle的魔法同步”的错误恢复机制。他的解决方案是：在语法中手工插入一条空规则的引用，该规则包含特定的、能够在必要时触发

错误恢复的动作。现在，**ANTLR 4**会在开始处和循环条件判定处自动插入同步检查，以避免激进的恢复机制。该方案详情如下：

子规则起始位置 在任意子规则的起始位置，语法分析器会尝试进行单词法符号移除。不过，和词法符号匹配不同的是，语法分析器不会尝试进行单词法符号补全。创建一个词法符号对**ANTLR**来说是很困难的，因为它必须猜测多个备选分支中的哪一个会最终胜出。

子规则的循环条件判定位置 如果子规则是一个循环结构，即 $(\dots)^*$ 或 $(\dots)^+$ ，在遇到错误时，语法分析器会尝试进行积极的恢复，使得自己留在循环内部。在成功地匹配到循环的某个备选分支之后，语法分析器会持续消费词法符号，直到发现满足下列条件之一的词法符号为止：

- (a) 循环的另一次迭代
- (b) 紧跟在循环之后的内容
- (c) 当前规则的重新同步集合中的元素

让我们先看看在子规则前的单词法符号移除。考虑**Simple**语法的 **classDef**规则中的**member**+循环结构。如果我们手误多输入了{，**member**+子规则会在进入**member**之前移除掉多余的那个词法符号，详见如图9-7所示的语法分析树。

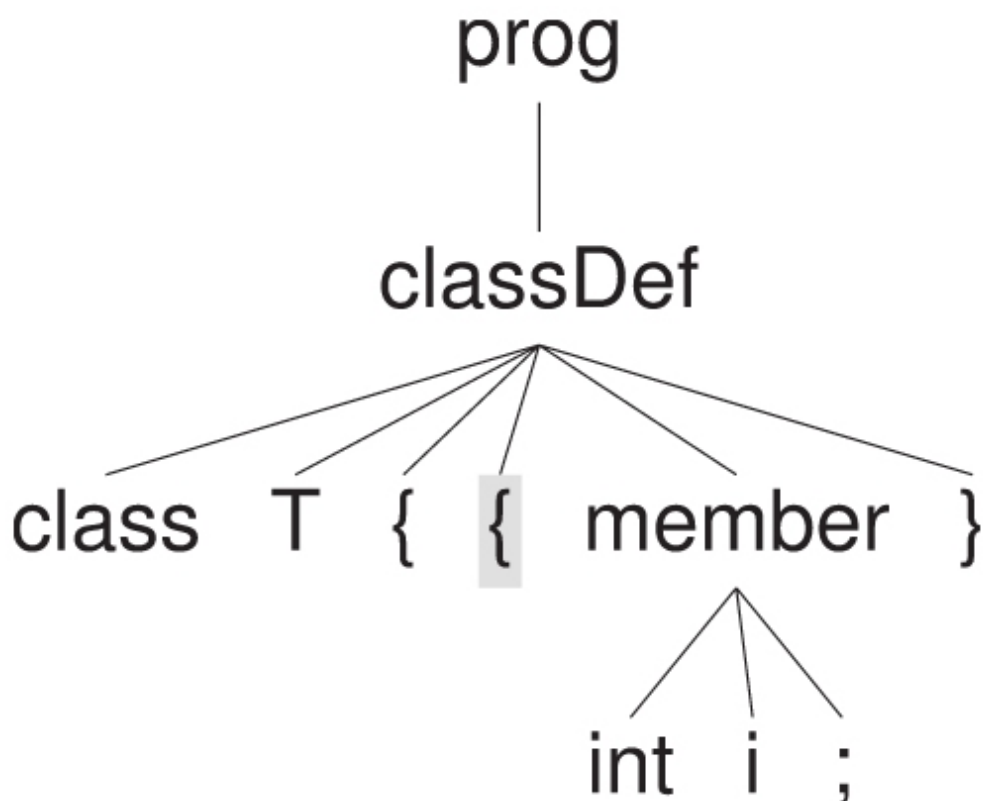


图9-7 移除多余词法符号的语法分析树

下面的命令行交互过程显示，ANTLR成功地进行错误恢复，因为它正确地识别出了变量*i*：

```

⇒ $ grun Simple prog
⇒ class T {{ int i; }
⇒ E0F
  < line 1:9 extraneous input '{' expecting 'int'
    var i
    class T

```

接下来，让我们试着输入一些真正杂乱无章的文本，看看member+循环能否从错误中恢复，继续寻找类成员。

```

⇒ $ grun Simple prog
⇒ class T {{
⇒   int x;
⇒   y;;;
⇒   int z;
⇒ }
⇒ EOf
  < line 1:9 extraneous input '{' expecting 'int'
    var x
    line 3:2 extraneous input 'y' expecting {'int', '}'
    var z
    class T

```

从中可知，语法分析器进行了重新同步，留在了循环内部，因为它识别出了变量`z`。语法分析器丢弃了`y; ; ;`，然后它发现了另外一个`member`的开始（即上面的条件c），于是它回到了`member`循环。如果输入文本不包含“`int z;` ”，语法分析器就会丢弃到`}`（上面的条件b）为止，然后退出循环。语法分析树高亮标记了被丢弃的词法符号，并显示出语法分析器仍然成功地将“`int z;` ”解释成了一个有效的类成员，如图9-8所示。

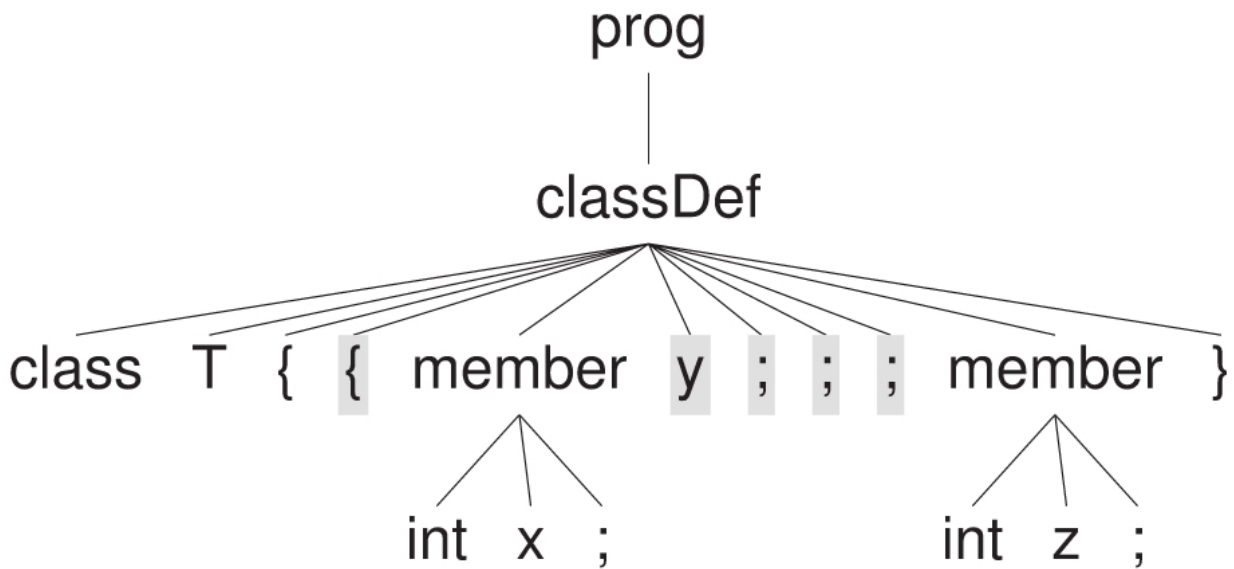


图9-8 被丢弃的词法符号被高亮标记的语法分析树

如果用户输入了一个非法的`member`，同时遗漏了类定义最后的`}`，我们不想希望语法分析器一直扫描到它发现`}`为止。如果这样的话，语法分析器的重新同步过程可能会丢弃后面的整整一个类定义，来寻找缺失的`}`。实际上，如果语法分析器发现了一个满足条件`c`的词法符号，它就会停止丢弃过程，如下所示：

```

⇒ $ grun Simple prog
⇒ class T {
⇒   int x;
⇒   ;
⇒ class U { int y; }
⇒ Eof
  < var x
    line 3:2 extraneous input ';' expecting {'int', '}'
    class T
    var y
    class U
  
```

从图9-9所示的语法分析树中，我们可以看出，语法分析器在它发现关键字class的时候就停止了重新同步过程。

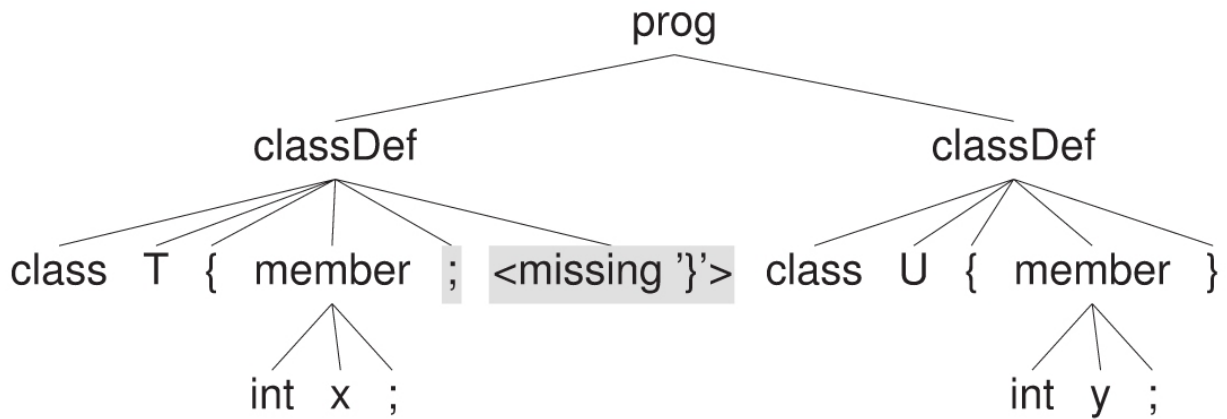


图9-9 停止了重新同步过程的语法分析树

除了词法符号匹配和子规则匹配中的失败，语法分析器还可能在匹配语义判定的时候失败。

4.捕获失败的语义判定

此时此刻，我们对语义判定的学习仅仅是浅尝辄止，不过，由于本章的主题是错误处理机制，在这里讨论语义判定失败时发生的事情是非常合适的。我们将在第11章中深入研究语义判定。目前，让我们暂时将语义判定看作断言。它们指定了一些必须在运行时为真的条件，以使得语法分析器能够通过这些条件的验证。如果一个判定结果为假，语法分析器会抛出一个FailedPredicateException异常，该异常会被当前

规则的`catch`语句捕获。语法分析器随即报告一个错误，并运行通用的同步-返回恢复机制。

下面让我们看一个使用语法判定来限制向量中整数数量的例子，它与4.4节“使用语义判定改变语法分析过程”部分中的语法非常相似。`ints`规则匹配最多`max`个整数。

```
errors/Vec.g4
vec4:  '[' ints[4] ']' ;
ints[int max]
locals [int i=1]
      :  INT ( ',' { $i++; } { $i <= $max } ? INT ) *
      ;
```

下列测试给出的整数过多，于是我们看到了一个错误消息，以及错误恢复的过程，在这个过程中，多余的逗号和整数被丢弃了：

```
⇒ $ antlr4 Vec.g4
⇒ $ javac Vec*.java
⇒ $ grun Vec vec4
⇒ [1,2,3,4,5,6]
⇒ EOF
  < line 1:9 rule ints failed predicate: { $i <= $max } ?
```

如图9-10所示的语法分析树显示，语法分析器在第五个整数处检测到了该错误。

fail选项接受两种参数：双引号包围的字符串常量或者一个可以得到字符串的动作。如果你希望在判定失败时执行一个函数，使用动作是极其方便的，只需在动作中调用该函数即可，例如{...}? <**fail**=
{failedMaxTest () }>。

关于使用语义判定来验证输入有效性这件事情，还有一些需要注意的地方。在上面的向量例子中，判定的强制性针对的是句法规则，所以抛出异常并尝试恢复是没问题的。但是，如果我们输入的结构在语法上是有效的，但是在语义上是无效的，这时，语义判定就不适用了。

假设存在一种语言，我们可以给一个变量赋予任何除零之外的值。这意味着“**assignment x=0;**”在语法上是有效的，但是在语义上是无效的。显然，这种情况下，我们需要向用户输出一个错误，但是不应该触发错误处理机制。“**x=0;**”在句法上是完全合法的。在某种意义上，语法分析器将会自动地从错误中“恢复”。下列简单语法展示了这个问题的处理方式：

```
errors/Pred.g4
assign
    : ID '=' v=INT {$v.int>0}? ';'
    {System.out.println("assign "+$ID.text+" to ");}
    ;
```

如果**assign**规则中的判定过程抛出了一个异常，同步-返回机制表现出的行为就会是丢弃判定后的“; ”。这种行为可能能够正常工作，但是我们面临的风险是不完美的重新同步。更好的解决方案是手工输出一个

错误，然后令语法分析器按照正确的语法继续进行匹配。所以，相比语义判定，我们应该使用一个带条件语句的动作。

```
{if ($v.int==0) notifyListeners("values must be > 0");}
```

现在，我们已经看到了所有会触发错误恢复机制的场景，需要指出的是，这种机制存在一个缺点。考虑到有时语法分析器在一次错误恢复的尝试中不会消费任何词法符号，这可能带来一个后果：整个恢复过程进入一个无限循环。如果语法分析器在恢复过程中没有消费任何词法符号，并且回到了相同的位置，那么我们会面临重新开始不消费词法符号的恢复过程的窘境。在下一节中，我们将会看到ANTLR是如何避免这个缺陷的。

5. 错误恢复机制的防护措施

ANTLR的语法分析器具有内置的防护措施，以保证错误恢复过程正常结束。如果我们在相同的语法分析位置，遇到了相同的输入情况，语法分析器会在尝试进行恢复之前强制消费一个词法符号。回到本章开头的简单Simple语法，让我们看一个能够触发防护措施的例子。如果我们在字段定义中加入一个多余的int，语法分析器就会检测到错误，从而尝试进行恢复。从下面的测试中我们可以看到，在正确的重新同步前，语法分析器会多次调用recover () 并尝试重新开始语法分析。


```

⇒ $ grun Simple prog
⇒ class T {
⇒   int int x;
⇒ }
⇒ E0F
  < line 2:6 no viable alternative at input 'intint'
    var x
    class T

```

如图9-11中的右侧语法分析树所示，classDef规则调用了三次member。

其中，第一个member没有匹配到任何内容，第二个member匹配到了多余的int。第三次匹配member的尝试正确地匹配到了“int x;”序列。

下面让我们详细分析这个过程。当语法分析器检测到第一个错误时，它正位于member规则中。

errors/Simple.g4

```

member
:   'int' ID ';'                                // 字段定义

    {System.out.println("var "+$ID.text);}
|   'int' f=ID '(' ID ')' '{' stat '}' // 方法定义
    {System.out.println("method: "+$f.text);}
;

```

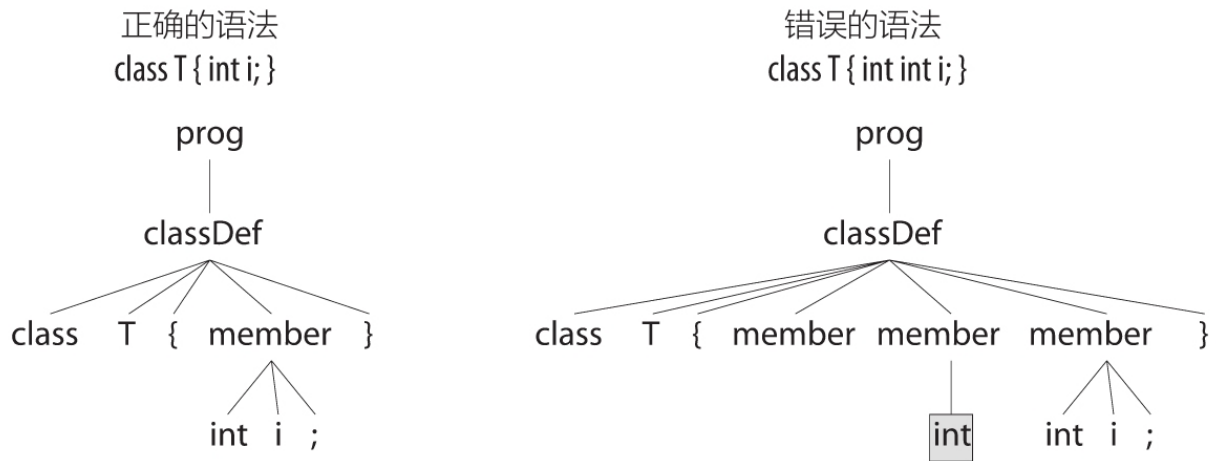


图9-11 正确和错误的语法对应的语法分析树

输入`int int`不适用`member`的任何一个备选分支，因此语法分析器执行了同步-返回错误恢复策略。它输出了第一条错误消息，然后开始消费词法符号，直到发现当前调用栈`[prog, classDef, member]`对应的重新同步集合中的词法符号为止。

因为语法中`classDef`+和`member`+循环的存在，计算重新同步集合的过程稍显复杂。在`member`的调用之后，语法分析器可能回到循环开头，再次匹配一个`member`，或者退出当前循环，匹配类定义尾部的`}`。在`classDef`的调用之后，语法分析器可能回到循环开头匹配另外一个类定义，或者简单地退出`prog`规则。因此，调用栈`[prog, classDef, member]`对应的重新同步集合就是`{'int', '}', 'class'}`。

此时，语法分析器发现，不需要消费词法符号就可以完成恢复，因为当前的输入词法符号`int`位于重新同步集合中。因此，它返回到了调用

者处：classDef规则的member+循环。该循环接着尝试匹配另一个类成员。不幸的是，由于它没有消费任何词法符号，语法分析器随即在返回member时再次检测到了错误（受errorRecovery标志位影响，它隐藏了重复的错误消息）。

在第二次错误的恢复中，语法分析器启用了防护措施，因为它在相同的语法分析位置遇到了相同的输入情况。在尝试重新同步之前，防护措施强制消费了一个词法符号。由于int位于重新同步集合中，它没有继续消费第二个词法符号。幸运的是，现在的情况正好是我们所期望的，因为语法分析器已经正确地完成了重新同步。接下来的三个词法符号代表一个有效的成员定义：“int x; ”。语法分析器的控制流再次从member回到了classDef中的循环。此时，我们第三次回到了member，不过，这一次语法分析已经能够顺利进行了。

这就是ANTLR自动错误恢复机制的全部细节。下面让我们学习一种手工的错误恢复机制，在某些情况下，它能够更好地完成恢复工作。

9.4 勘误备选分支

一些语法错误十分常见，以至于对它们进行特殊处理是值得的。例如，开发者经常在嵌套的函数调用后写错括号的数量。为了对这些情况进行特殊处理，我们只需增加一些备选分支，匹配这些常见错误即

可。下面的语法识别单参数的函数调用，其中参数中可能包含嵌套的括号。fcall规则具有两个所谓的勘误备选分支（error alternative）。

errors/Call.g4

```
stat:  fcall ';' ;
fcall
    :   ID '(' expr ')'
    |   ID '(' expr ')' ')' {notifyErrorListeners("Too many parentheses");}
    |   ID '(' expr          {notifyErrorListeners("Missing closing ')'");}
    ;

expr:  '(' expr ')'
    |   INT
    ;
```

这些勘误备选分支会给ANTLR自动生成的语法分析器带来少量的额外工作，但是不会对它形成干扰。和其他的备选分支一样，只要输入文本与之相符，语法分析器就会匹配到它们。在下面的例子中，我们首先输入了一个合法的函数调用，随后输入了一些匹配勘误备选分支的文本。

```
⇒ $ antlr4 Call.g4
⇒ $ javac Call*.java
⇒ $ grun Call stat
⇒ f(34);
⇒ EoF
⇒ $ grun Call stat
⇒ f((34));
⇒ EoF
  < line 1:6 Missing closing ')'
⇒ $ grun Call stat
⇒ f((34)));
⇒ EoF
  < line 1:8 Too many parentheses
```

迄今为止，我们已经学习了相当多错误处理方面的知识，它们包括ANTLR语法分析器能够产生的错误消息，以及语法分析器在多种情况下的错误恢复机制。我们也看到了自定义错误消息和将错误消息转发到不同错误监听器的方法。上述所有功能都由一个对象封装和控制，该对象指定了ANTLR的错误处理策略。在下一节中，我们将会详细了解该策略的细节，以便深入学习如何自定义语法分析器对错误的处理行为。

9.5 修改ANTLR的错误处理策略

默认的错误处理机制表现出色，不过我们还是会遇到一些非典型的、需要修改默认机制的场景。首先，我们希望关闭某些默认的错误处理功能，它们会带来额外的运行负担。其次，我们可能希望语法分析器在遇到第一个语法错误时就退出。这种情况的例子是，当处理类似bash的命令行输入时，从错误中恢复是毫无意义的。我们不能一意孤行地执行有风险的命令，因此语法分析器可以一遇到问题就退出。

欲探究错误处理策略，不妨查看一下ANTLRErrorStrategy接口及实现类DefaultError-Strategy。该类完成了全部默认错误处理工作。例如，下面的语句是每个ANTLR自动生成的规则函数中的catch中的内容：

```
_errHandler.reportError(this, re);  
_errHandler.recover(this, re);
```

`_errHandler`是一个指向`DefaultErrorStrategy`实例的变量。`reportError ()`方法和`recover ()`方法实现了错误的报告和同步-返回功能。`reportError ()`方法根据抛出的异常类型，将报告错误的职责委托给了另外三个方法之一。

对于之前提到的第一种非典型场景：减少错误处理机制给语法分析器带来的运行负担。请看下面的代码，它是ANTLR根据Simple语法中的`member+`子规则自动生成的：

```
_errHandler.sync(this);
_la = _input.LA(1);
do {
    setState(22); member();
    setState(26);
    _errHandler.sync(this);
    _la = _input.LA(1);
} while ( _la==6 );
```

在某些程序中，可以假定输入在句法上是正确的，例如网络协议。在这种情况下，我们最好避免错误检查和恢复带来的负荷。我们可以通过以下方法达到这个目的：继承`DefaultErrorStrategy`类，并使用一个空方法覆盖`sync ()`。Java编译器通常会在后续的优化过程中将`_errHandler.sync (this)`调用内联化，并执行无用代码消除。在下一个例子中，我们将会看到如何令语法分析器采取不同的错误处理策略。

另外一种非典型场景是令语法分析器在第一个语法错误处退出。为了达到这个目的，我们需要覆盖三个关键方法，详情如下：

```
errors/BailErrorStrategy.java
import org.antlr.v4.runtime.*;

public class BailErrorStrategy extends DefaultErrorStrategy {
    /** 不从异常 e 中恢复，而是用一个通用的
     * RuntimeException 包装它，这样它
     * 就不会被规则函数中的 catch 语句捕获。
     * 异常 e 是生成的 RuntimeException 的 cause 成员。
     */

    @Override

    public void recover(Parser recognizer, RecognitionException e) {
        throw new RuntimeException(e);
    }

    /** 确保不会试图执行行内恢复，如果语法分析器
     * 成功地进行了恢复，它就不会抛出一个异常
     */
    @Override
    public Token recoverInline(Parser recognizer)
        throws RecognitionException
    {
        throw new RuntimeException(new InputMismatchException(recognizer));
    }

    /** 确保不会试图从子规则的问题中恢复 */
    @Override
    public void sync(Parser recognizer) { }
}
```

出于测试的目的，我们可以复用一些样例代码。除了创建和启动语法分析器外，我们还需要创建一个新的BailErrorStrategy实例，并且令语法分析器使用它来替代默认的错误处理策略。

```
errors/TestBail.java
parser.setErrorHandler(new BailErrorStrategy());
```

随后，我们应当令它在第一个词法错误处报错并退出。要达到这个目的，只需覆盖Lexer类中的recover方法即可。

```
errors/TestBail.java
public static class BailSimpleLexer extends SimpleLexer {
    public BailSimpleLexer(CharStream input) { super(input); }
    public void recover(LexerNoViableAltException e) {
        throw new RuntimeException(e); // 报错退出
    }
}
```

让我们先在输入文本的开头插入一个#字符，以构造一个词法错误。可见，词法分析器抛出了一个异常，接管了主程序中的控制流。

```
⇒ $ antlr4 Simple.g4
⇒ $ javac Simple*.java TestBail.java
⇒ $ java TestBail
⇒ # class T { int i; }
⇒ EOF
< line 1:1 token recognition error at: '#'
Exception in thread "main"
java.lang.RuntimeException: LexerNoViableAltException('#')
    at TestBail$BailSimpleLexer.recover(TestBail.java:9)
    at org.antlr.v4.runtime.Lexer.nextToken(Lexer.java:165)
    at org.antlr.v4.runtime.BufferedTokenStream.fetch(BufferedTokenStream.java:139)

at org.antlr.v4.runtime.BufferedTokenStream.sync(BufferedTokenStream.java:133)
at org.antlr.v4.runtime.CommonTokenStream.setup(CommonTokenStream.java:129)
at org.antlr.v4.runtime.CommonTokenStream.LT(CommonTokenStream.java:111)
at org.antlr.v4.runtime.Parser.enterRule(Parser.java:424)
at SimpleParser.prog(SimpleParser.java:68)
at TestBail.main(TestBail.java:23)
...
```


同时，语法分析器在第一个语法错误（本例中的类名缺失）处就退出了。

```
⇒ $ java TestBail
⇒ class { }
⇒ EOF
⚡ Exception in thread "main" java.lang.RuntimeException:
    org.antlr.v4.runtime.InputMismatchException
    ...
```

为展示ANTLRErrorStrategy接口的灵活性，让我们通过一个例子来圆满结束本章的学习：修改语法分析器的错误报告策略。如果希望修改标准的错误消息“在输入X处没有可行的备选分支”，我们可以覆盖reportNoViableAlternative () 方法，将错误消息改成其他内容。

```
errors/MyErrorStrategy.java
import org.antlr.v4.runtime.*;
public class MyErrorStrategy extends DefaultErrorStrategy {
    @Override
    public void reportNoViableAlternative(Parser parser,
                                         NoViableAltException e)
        throws RecognitionException
    {
        // ANTLR 基于语法生成的语法分析器是 Parser 的子类，
        // Parser 类继承了 Recognizer 类
        // 方法参数 parser 指向检测到错误的语法分析器
        String msg = "can't choose between alternatives"; // 自定义的非标准消息
        parser.notifyErrorListeners(e.getOffendingToken(), msg, e);
    }
}
```

不过，请记住，如果我们需要的仅仅是改变错误消息输出的位置，我们可以像9.2节做的那样，指定一个ANTLRErrorListener。欲了解如何

完全覆盖ANTLR生成的异常捕获代码，请阅读15.3节“捕获异常”部分。

在本章中，我们介绍了ANTLR中重要的错误报告和恢复机制的全部细节。利用ANTLRErrorListener和ANTLRErrorStrategy接口，我们能够非常灵活地指定错误消息的输出位置、错误消息的内容以及语法分析器从错误中恢复的方法。

在下一章中，我们会学习如何在语法中直接嵌入被称为动作（action）的代码片段。

第10章 属性和动作

在之前的学习中，我们的程序逻辑代码都是与语法分析树遍历器分离的，这意味着我们的代码总是在语法分析完成之后执行。在接下来的几章中我们可以看到，一些语言类应用程序需要在语法分析的过程中执行自身的逻辑代码。为了达到这个目的，我们需要一种手段，将代码片段——称为动作——直接注入ANTLR生成的代码中。本章的第一个目标是，学习如何在语法分析器和词法分析器中嵌入动作，并弄清楚我们可以在这些动作中放置哪些内容。

请记住，通常我们应当避免将语法和应用程序的逻辑代码纠缠在一起。不包含动作的语法更易阅读，也不会绑定到特定的目标语言和程

序上。尽管如此，内嵌的动作仍然是有用的，原因有如下三个：

- 简便：有时，使用少量的动作，避免创建一个监听器或者访问器会使事情变得更加简单。

- 效率：在资源紧张的程序中，我们可能不想把宝贵的时间和内存浪费在建立语法分析树上。

- 带判定的语法分析过程：在某些罕见情况下，我们必须依赖从之前的输入流中获取的数据才能正常地进行语法分析过程。一些语法需要建立一个符号表，以便在未来根据情况（例如一个标识符是类型还是方法）差异化地识别输入的文本。我们已经在第11章中探究过这样的例子。

动作就是使用目标语言（即ANTLR生成的代码的语言）编写的、放置在`{...}`中的任意代码块。我们可以在动作中编写任意代码，只要它们是合法的目标语言语句。动作的典型用法是操纵词法符号和规则引用的属性（`attribute`）。例如，我们可以读取一个词法符号对应的文本或者整个规则匹配的文本。通过从词法符号和规则引用中获取的数据，我们就可以打印结果或者执行任意计算。规则允许参数和返回值，因此我们可以在规则之间传递数据。

我们将会通过三个例子来学习编写语法中的动作。第一，我们会编写一个计算器，它的功能与7.4节中的计算器相同。第二，我们会为CSV

语法（见6.1节）增加一些动作，以此来探索规则和词法符号的属性。在第三个例子中，我们将会为一门在运行期才能确定关键字的语言编写一个语法，以此来学习词法规则中的动作。

是动手的时候了，下面让我们从一个基于动作的计算器实现开始。

10.1 使用带动作的语法编写一个计算器

让我们通过回顾4.2节中的表达式语法来学习编写动作。在该节中，我们利用访问器编写了一个能够对表达式求值的计算器，如下所示：

```
actions/t.expr
```

```
x = 1
```

```
x
```

```
x+2*3
```

我们的目标是在不使用访问器，甚至不建立语法分析树的前提下，重新编写一个功能相同的计算器。此外，我们还会利用一个小技巧使其具备交互功能，这意味着我们会在敲回车时获得结果，而非在输入结束后。相比之下，之前的所有示例都是先读取完整的输入文本，然后处理生成的语法分析树。

通过本节，我们会习得以下技能：将生成的语法分析器放入包中、定义语法分析器的字段和方法、在备选分支中插入动作、标记语法元素以便在动作中使用，以及定义规则的返回值。

1.在语法规则之外使用动作

在语法规则之外，我们希望将两种东西注入自动生成的语法分析器和词法分析器：`package/import`语句以及类似字段和方法这样的类成员。

下面是一份理想化的代码生成模板，它展示了在语法分析器这样的自动生成的代码中，我们希望注入代码片段的位置。

```
<header>
public class <grammarName>Parser extends Parser {
    <members>
    ...
}
```

我们可以在语法中使用`@header{...}`来指定一段header动作代码，使用`@members{...}`向生成的代码中注入字段或者方法。在一个联合了文法和词法的语法中，这些具名的动作会同时应用于语法分析器和词法分析器（ANTLR选项-`package`允许我们直接设定包名，而无需使用header动作）。如果需要限制一段动作代码只出现在语法分析器或者词法分析器中，我们可以使用`@parser: : name`或者`@lexer: : name`。下面让我们看看我们的计算器是如何使用上述特性的。和之前相同，计算器使用到的表达式语法以一个语法声明开始，不过，现在我们打算将所有生成的代码声明于一个特定的Java包中。此外，我们还需要导入一些标准的Java工具类。

```
actions/tools/Expr.g4
grammar Expr;

@header {
package tools;
import java.util.*;
}
```

之前的计算器的EvalVistor类有一个存储键值对的memory字段，它用于实现变量的赋值和引用。在本次实现中，我们会将这个字段放入members功能里。为避免语法显得凌乱，我们还定义了一个eval () 方法，用于对两个操作数执行相关操作。下面是完整的members动作：

```
actions/tools/Expr.g4
@parser::members {
    /** "memory" 字段用于存储变量 / 变量值对 */
    Map<String, Integer> memory = new HashMap<String, Integer>();

    int eval(int left, int op, int right) {
        switch ( op ) {
            case MUL : return left * right;
            case DIV : return left / right;
            case ADD : return left + right;
            case SUB : return left - right;
        }
        return 0;
    }
}
```

完成上述定义之后，让我们看看如何在规则内的动作中使用这些类成员。

2.在规则中嵌入动作

在本节中，我们将会学习在语法中嵌入动作，这些动作可以生成输出、更新数据结构，或者设置规则的返回值。我们还会看到ANTLR是

如何将规则的参数、返回值，以及规则调用的其他属性包装成一个 `ParserRuleContext` 子类的实例的。

(1) 基础知识

`stat` 规则用于识别表达式、变量赋值语句和空行。因为我们在发现空行时什么都不做，所以 `stat` 规则只需要两个动作。

```
actions/tools/Expr.g4
stat:  e NEWLINE      {System.out.println($e.v);}
      | ID '=' e NEWLINE {memory.put($ID.text, $e.v);}
      | NEWLINE
      ;
```

动作被执行的时机是它前面的语法元素之后、它后面的语法元素之前。在本例中，动作出现在备选分支的末尾，因此它们会在语法分析器匹配到整个语句之后被执行。当 `stat` 发现一个后面跟着 `NEWLINE` 的表达式时，它应当打印出该表达式的值；当 `stat` 发现一个变量赋值语句时，它就应当将该键值对存储到 `memory` 字段中。

这些动作代码中唯一陌生的语法是 `$e.v` 和 `$ID.text`。通常，`$x.y` 是指元素 `x` 的 `y` 属性，其中 `x` 可以是词法符号引用或者规则引用。在这里，`$e.v` 指的是调用规则 `e` 的返回值（稍后我们会看到为什么它被称为 `v`）。

`$ID.text` 指的是 `ID` 词法符号匹配到的文本。

如果 `ANTLR` 无法识别 `y` 属性，它就不会转换该属性。在本例中，`text` 是一个词法符号的已知属性，所以 `ANTLR` 将它转换为了 `getText()`。我

们还可以使用`$ID.getText()`来达到相同效果。有关规则和词法符号的属性的完整列表，请参阅15.4节。

回到规则`e`，让我们来看看内嵌在其中的动作。我们的初衷是通过直接向语法中插入代码片段，即动作，来模拟`EvalVisitor`的功能。

```
actions/tools/Expr.g4
e returns [int v]
: a=e op=('*' | '/') b=e  {$v = eval($a.v, $op.type, $b.v);}
| a=e op=('+' | '-') b=e  {$v = eval($a.v, $op.type, $b.v);}
| INT                      {$v = $INT.int;}
| ID
{
    String id = $ID.text;
    $v = memory.containsKey(id) ? memory.get(id) : 0;
}
| '(' e ')'                {$v = $e.v;}⊖
;
```

（注：根据原书勘误表，此处原文有误，已修正。——译者注）

这个例子中有许多引人入胜的细节。我们发现的第一个细节是它指定了一个整数类型的返回值`v`。这就是之前`stat`的动作中引用`$e.v`的原因。`ANTLR`的返回值和`Java`的返回值的差异在于，我们需要为它们命名，并且可以有多个返回值。

接下来，我们看到了规则引用`e`和运算符子规则上的标记，如`op=`

`('*' | '/')`。标记可以指向一个词法符号，也可以指向在匹配词法符号或规则过程中生成的`ParserRuleContext`对象。

在详细分析动作的内容之前，有必要了解一下ANTLR存储诸如返回值和标记这样的信息的位置。在进行源代码级别的调试时（source-level debug），拥有这些知识会使得ANTLR自动生成的代码更易理解。

（2）将一切打包成一个规则上下文对象

在2.4节中，我们已经了解到，ANTLR通过规则上下文对象（rule context object）来实现语法分析树的节点。每次规则调用都会新建并返回一个规则上下文对象，它存储了相应规则在输入流的特定位置上识别工作的所有重要信息。例如，规则e新建并返回EContext对象。

```
public final EContext e(...) throws RecognitionException {...}
```

自然地，规则上下文对象非常适合放置与特定规则相关的数据实体。EContext的第一部分如下所示：

```
public static class EContext extends ParserRuleContext {  
    public int v;           // 规则 e 的返回值，源于 "returns [int v]"  
    public EContext a;      // （递归的）规则引用 e 上的标记 a  
    public Token op;        // 类似 ('*' | '/') 的运算符子规则上的标记  
    public EContext b;      // （递归的）规则引用 e 上的标记 b  
    public Token INT;       // 第三个备选分支引用的 INT  
    public Token ID;        // 第四个备选分支引用的 ID  
    public EContext e;      // e 的调用过程对应的上下文对象的引用  
    ...  
}
```

标记总是会成为规则上下文对象的成员，但是ANTLR并不总是为类似ID、INT和e的备选分支元素生成字段。ANTLR只有在它们被语法中的

动作引用时才为它们生成字段（例如e中的动作）。ANTLR会尽可能地减少上下文对象中字段的数量。

现在，我们的计算器的各部分都已就绪，让我们一起分析一下规则e的备选分支中动作的内容。

（3）计算返回值

e中的所有动作都通过赋值语句“\$v=...;”来设置返回值。该语句虽然设置了返回值，但是并不会导致对应的规则函数返回（不要在动作中使用return语句，它会使语法分析器崩溃）。下面是开头两个备选分支使用的动作：

```
$v = eval($a.v, $op.type, $b.v);
```

这段代码计算子表达式的值并将其赋给了e的返回值。eval（）方法的参数是两个e引用的返回值\$a.v和\$b.v，以及当前备选分支匹配到的运算符类型\$op.type。\$op.type必然是某个算术运算符的词法符号类型。注意我们可以重复使用同一个标记（只要它们指向相同类型的对象）。因此，第二个备选分支重复使用了标记a、b和op。

第三个备选分支的动作中使用\$INT.int来访问INT词法符号匹配到的文本对应的整数。它仅仅是Integer.valueOf（\$INT.text）的简写。这些内

嵌的动作比等价的访问器方法`visitInt()`要简单得多（代价是使程序的逻辑代码和语法纠缠在了一起）。

```
tour/EvalVisitor.java
/** INT */
@Override
public Integer visitInt(LabeledExprParser.IntContext ctx) {
    return Integer.valueOf(ctx.INT().getText());
}
```

第四个备选分支识别一个变量的引用，如果在此之前它的值已经被存储过，就将`e`的返回值设置为该变量在`memory`中的值。这段动作代码使用了Java的`?:`运算符，不过我们也能轻易地将它改写成`if-else`的形式。我们可以在动作中放入任何东西，只要它们能在Java方法中正常工作即可。

最后一个备选分支中的`$v=$e.v`；动作将返回值设为括号中的表达式的值。在这里，我们仅仅是传递了一个返回值而已。（3）的值就是3。

以上就是全部的语法和动作。下面让我们学习编写计算器的交互部分。

（4）编写一个交互式的计算器

在探索交互工具的细节之前，让我们先熟悉一下该语法和`Calc.java`的构建和测试过程。由于在`header`动作中加入了语句“`package tools;`”，我们需要将生成的Java代码放入一个名为`tools`的目录（它反映了Java标准

下的包名和目录结构之间的关系)。这意味着, 我们需要在tools目录中运行ANTLR, 或者在它的上级目录中指定路径tools/Expr.g4来运行。

```
$ antlr4 -no-listener tools/Expr.g4 # 在 tools 目录下生成不带监听器的语法分析器
$ javac -d . tools/*.java         # 编译且将生成的 .class 文件放在 tools 下
```

下面我们使用Calc的全限定类名来试一下。

```
⇒ $ java tools.Calc
⇒ x = 1
⇒ x
  < 1
⇒ x+2*3
  < 7
⇒ EOF
```

你会注意到, 当你敲回车的时候, 计算器立刻给出了结果。在默认情况下, ANTLR会读取全部的输入(通常是读入一个巨大的缓冲区), 为达到上述目的, 我们必须将输入文本一行一行地传递给它, 以使得这个过程变成交互式的。每行代表一个完整的表达式(如果需要处理可以分为多行的表达式, 参见12.2节“有趣的Python换行符”部分)。下面的main ()方法是我们获得第一个表达式的方式:

actions/tools/Calc.java

```
BufferedReader br = new BufferedReader(new InputStreamReader(is));
String expr = br.readLine();           // 获取第一个表达式
int line = 1;                          // 跟踪输入的表达式行号
```

为在不同的表达式之间共享memory字段的值, 我们需要用同一个语法分析器实例处理所有的输入行。

```
actions/tools/Calc.java
```

```
ExprParser parser = new ExprParser(null); // 共享同一个语法分析器的实例  
parser.setBuildParseTree(false);          // 不需要建立语法分析树
```

当我们读入一行时，我们需要新建一个词法符号流，将其传给共享的语法分析器。

```
actions/tools/Calc.java
```

```
while ( expr!=null ) {                // 当多于一个表达式时  
    // 为每行（每个表达式）新建一个词法分析器和词法符号流  
  
    ANTLRInputStream input = new ANTLRInputStream(expr+"\n");  
    ExprLexer lexer = new ExprLexer(input);  
    lexer.setLine(line);              // 通知词法分析器输入的位置  
    lexer.setCharPositionInLine(0);  
    CommonTokenStream tokens = new CommonTokenStream(lexer);  
    parser.setInputStream(tokens);    // 用新的词法符号流通知语法分析器  
    parser.stat();                    // 开始语法分析过程  
    expr = br.readLine();             // 检查下一行是否存在  
    line++;  
}
```

现在，我们已经掌握了编写交互式工具的方法，并且清楚了如何编写和使用内嵌动作。我们的计算器使用一段**header**动作指定了包名，同时使用**members**动作为语法分析器定义了两个类成员。我们将规则内的动作用作处理词法符号和规则属性的函数，从而能够计算和返回子表达式的值。在下一节中，我们会看到更多的属性，了解更多放置动作的可行位置。

10.2 访问词法符号和规则的属性

让我们以6.1节中的CSV语法为基础，学习一些与动作相关的特性。我们会编写一个程序，解析并打印CSV文件中的数据，它会为每行生成一个从列名到字段值的Map。我们的目的是学习更多有关规则动作和属性的知识。

首先，让我们看看如何使用**locals**区域（**section**）定义局部变量。经过定义参数和返回值后，**locals**区域中的声明就会成为规则上下文对象的字段。由于我们在每次规则调用时都会获得一个新的规则上下文，可以预料，我们同时也获得了**locals**的一份新拷贝。下面这个版本的**file**规则带有参数，包含很多有趣的细节，不过，让我们首先重点关注一下**locals**到底可以做什么。

```
actions/CSV.g4
```

```
/** 由规则 "file : hdr row+ ;" 衍生而来 */
file
locals [int i=0]
    : hdr ( rows+=row[$hdr.text.split(",")] {$i++;} )+
    {
        System.out.println($i+" rows");
        for (RowContext r : $rows) {
            System.out.println("row token interval: "+r.getSourceInterval());
        }
    }
    ;
```

file规则定义了一个局部变量*i*，并且使用动作代码*\$i++*来统计当前输入的行数。引用局部变量时请不要忘了\$前缀，否则编译器会报告变量未定义的错误。ANTLR将*\$i*转换成`_localctx.i`；在**file**规则对应的规则函数中，实际上是不存在局部变量*i*的。

接下来，让我们看看对row规则的调用。规则调用row[\$hdr.text.split
(", ")]显示，我们使用方括号而非圆括号来向规则传递参数（圆括
号已经被ANTLR的子规则语法占用了）。参数表达式\$hdr.text.split
(", ") 将hdr规则匹配到的文本切分为一组row规则所需的字符串。

让我们分别理解这件事情。\$hdr是对唯一的hdr规则调用的引用，它指
向本次调用的HdrContext对象。在本例中，我们无需对hdr规则引用进
行标记（如h=hdr）的原因是\$hdr是独一无二的。因此，\$hdr.text就是标
题行匹配到的文本。我们使用标准的Java方法String.split () 将逗号分
隔的标题列切分为一组字符串。我们稍后会看到row规则接收一个字符
串数组作为参数。

对row的调用也引入了一种新的标记，即+=而非=标记符。相比之下，=
用于跟踪单个值，而这里的标记rows是所有的row调用返回的
RowContext对象的List。在打印出rows的数量后，file规则中最后的动
作代码通过一个循环遍历了所有的RowContext对象。在循环的每次迭
代中，它都打印出row规则调用匹配到的词法符号的索引值范围（使用
getSourceInterval () 方法）。

循环使用了r，而非\$r，这是因为r是一段Java代码中的局部变量。
ANTLR只能看到locals关键字定义的局部变量，而无法看到用户编写的
任意内嵌动作中的局部变量。它们之间的差别在于，file规则对应的语
法分析树节点只会定义字段i，而不会定义字段r。

现在转到**hdr**规则，在该规则中，我们仅仅打印出标题行的内容。我们可以通过**\$hdr.text**来完成这项工作，它就是**row**规则引用匹配到的文本。另外，我们也可以直接用**\$text**获得当前的规则匹配到的文本。

```
actions/CSV.g4
```

```
hdr : row[null] {System.out.println("header: '"+$text.trim()+"'");} ;
```

在本例中，它同时也是**row**规则匹配到的文本，因为它们二者包含的内容是相同的。

现在让我们使用**row**规则中的动作，将每行数据转换成一个从列名到列值的**Map**。首先，**row**接收一组列名作为参数，返回一个**Map**。其次，为了在列名组成的数组中移动，我们需要一个局部变量**col**。在解析该行数据之前，我们需要初始化返回的**Map**，另外，让我们再找个乐子，**row**结束后打印出**Map**里的值。上述内容组成了该规则的头部。

```
actions/CSV.g4
```

```
/* 由规则 "row : field (',' field)* '\r'? '\n' ;" 衍生而来 */
row[String[] columns] returns [Map<String,String> values]
locals [int col=0]
@init {
    $values = new HashMap<String,String>();
}
@after {
    if ($values!=null && $values.size()>0) {
        System.out.println("values = "+$values);
    }
}
```

init动作发生在对应规则匹配过程开始之前，无论它有多少个备选分支。同样，**after**动作发生在对应规则的备选分支之一完成匹配之后。在

这个例子中，我们将打印语句置于row规则的最外层备选分支的末尾，以阐明after动作的功能。

当一切就绪后，我们就可以提取数据并填充该Map了。

actions/CSV.g4

```
// 继续上文的 row 规则
: field
{
  if ($columns!=null) {
    $values.put($columns[$col++].trim(), $field.text.trim());
  }
}
( ',' field
{
  if ($columns!=null) {
    $values.put($columns[$col++].trim(), $field.text.trim());
  }
}
)* '\r'? '\n'
;
```

这段动作的主要部分通过\$values.put (...) 将列名对应字段的值存储了结果map中。这个方法的第一个参数是这样得到的：获得列名，将索引值增一，然后使用\$columns[\$col++].trim () 移除列名两侧的空白。第二个参数通过\$field.text.trim () 移除掉最近一次匹配到的字段文本两侧的空白（row中的两段动作代码是完全相同的，所以最好将它们重构为members动作中的一个方法）。

CSV.g4中的其他内容我们都已经很熟悉了，所以这里不再赘述，直接进行它的构建和测试过程。因为grun的存在，我们无须为其编写特殊的测试，可以只生成语法分析器并编译之。

```
$ antlr4 -no-listener CSV.g4 # 这次我们仍然不使用监听器
$ javac CSV*.java
```

下面是我们使用的CSV数据:

```
actions/users.csv
User, Name, Dept
parrt, Terence, 101
tombu, Tom, 020
bke, Kevin, 008
```

下面是输出结果:

```
$ grun CSV file users.csv
header: 'User, Name, Dept'
values = {Name=Terence, User=parrt, Dept=101}
values = {Name=Tom, User=tombu, Dept=020}
values = {Name=Kevin, User=bke, Dept=008}
3 rows
row token interval: 6..11
row token interval: 12..17

row token interval: 18..23
```

hdr规则打印出了上面的第一行输出，然后三次对**row**的调用打印出了三行**values=...**。此时，程序的控制流程回到了**file**规则，它的动作打印出了总行数以及每行数据包含的词法符号位置范围。

至此，我们已经掌握了内嵌动作的用法，无论是在规则内部还是规则外部。此外，我们还学习了些许有关规则属性的知识。不过，计算器和CSV处理器的例子都只在文法规则中使用了动作。实际上，动作在

词法规则中也可以大放光彩。我们将会在下一节中通过处理巨量和动态的关键字集合，来学习与之相关的知识。

10.3 识别关键字不固定的语言

为探究内嵌在词法规则中的动作的相关知识，让我们为一门虚拟的、关键字会动态变化（每次运行都不同）的编程语言编写一份语法。这件事情听上去不可思议，但确实是可能的。例如，Java 5新增了一个关键字`enum`，因此同一个编译器必须能够根据“`-version`”选项动态地开启和关闭它。

也许，更常见的应用是处理拥有巨量关键字集合的语言。我们可以令词法分析器分别匹配所有的关键字（作为独立的规则），也可以编写一条ID规则作为分发器，然后在一个关键字列表中查找该规则匹配到的标识符。如果词法分析器发现该标识符是一个关键字，我们可以它的词法符号类型从原先通用的ID类型改成相应的关键字类型。

在着手实现ID规则和关键字查找机制之前，让我们先来编写包含关键字引用的语句规则。

actions/Keywords.g4

```
stat:  BEGIN stat* END
      |  IF expr THEN stat
      |  WHILE expr stat
      |  ID '=' expr ';'
      ;

expr:  INT | CHAR ;
```

虽然ANTLR会隐式地为每个关键字（BEGIN、END等）定义一个词法符号类型。但是，它会警告我们，这些词法符号类型没有对应的词法定义。

```
$ antlr4 Keywords.g4
warning(125): Keywords.g4:31:8: implicit definition of token BEGIN in parser
...
```

为关闭这个警告，我们需要进行显式定义。

```
actions/Keywords.g4
// 显式定义关键字的词法符号类型，避免隐式定义产生的警告
tokens { BEGIN, END, IF, THEN, WHILE }
```

在生成的KeywordsParser中，ANTLR定义的词法符号类型像是这样：

```
public static final int ID=3, BEGIN=4, END=5, IF=6, ... ;
```

既然我们已经完成了词法符号类型的定义，让我们看看这份语法的声明和header动作，它导入了Map和HashMap。

```
actions/Keywords.g4
grammar Keywords;
@lexer::header { // 只在词法分析器中放置这个 header，在语法分析器中不放置它
import java.util.*;
}
```

我们将会使用一个Map存放从关键字到其整数词法符号类型的映射作为关键字表。另外，我们还使用内联的Java实例初始化语句（内层的花括号中的代码）定义了一个Map。

actions/Keywords.g4

```
@lexer::members { // 只在词法分析器中放置这个成员
Map<String,Integer> keywords = new HashMap<String,Integer>() {{
    put("begin", KeywordsParser.BEGIN);
    put("end",    KeywordsParser.END);
    put("if",     KeywordsParser.IF);
    put("then",   KeywordsParser.THEN);
    put("while",  KeywordsParser.WHILE);
}};
}
```

这一切准备就绪之后，让我们开始进行之前做过多次的匹配标识符的工作，不过，这次我们使用了一段动作代码来将词法符号类型设置为恰当的值。

actions/Keywords.g4

```
ID : [a-zA-Z]+
    {
        if ( keywords.containsKey(getText()) ) {
            setType(keywords.get(getText())); // 重置词法符号类型
        }
    }
;
```

在这里，我们使用了**Lexer**类的**getText ()**方法来获取当前词法符号的文本内容。我们根据它的文本内容来确定它是否存在于**keywords**中。如果存在，那么我们就将该词法符号的类型从**ID**重置为相应关键字的词法符号类型。

在和词法分析器打交道时，我们需要清楚如何修改一个词法符号的文本内容。它可以用于剥离字符常量或者字符串常量两侧的引号。通常，一个语言类应用程序只需要引号中的文本。下面是使用**setText ()**覆盖一个词法符号中的文本的方法：

```
actions/Keywords.g4
```

```
/** 将 3 个字符的 'x' 输入序列转换成字符串 x */  
CHAR:  '\'' . '\'' {setText( String.valueOf(getText().charAt(1)) )}; ;
```

如果我们想要做一些真正疯狂的事情，我们甚至可以用`setToken()`方法指定词法分析器返回的`Token`对象。这是一种返回自定义的词法符号的方式。另外一种方式是覆盖`Lexer`的`emit()`方法。

至此我们已经准备就绪，可以开始体验这门微型语言了。我们期望的行为是它能将关键字和常规的标识符区分开，换句话说，“`x=34;`”应是合法的，但是“`if=34;`”不是，因为`if`是一个关键字。让我们运行`ANTLR`，编译生成的代码，然后使用合法的赋值语句测试它。

```
⇒ $ antlr4 -no-listener Keywords.g4  
⇒ $ javac Keywords*.java  
⇒ $ grun Keywords stat  
⇒ x = 34;  
⇒ EOF
```

没问题，没有任何错误发生。但是，当试图输入将`if`用作标识符的赋值语句时，语法分析器给出了一个语法错误。同时，它也能接受合法的`if`语句而不输出任何错误。

```
⇒ $ grun Keywords stat  
⇒ if = 34;  
⇒ EOF  
⌞ line 1:3 extraneous input '=' expecting {CHAR, INT}  
   line 2:0 mismatched input '<EOF>' expecting THEN  
⇒ $ grun Keywords stat  
⇒ if 1 then i = 4;  
⇒ EOF
```

如果你很不幸，正在为一门在某些上下文中允许将关键字当作标识符的编程语言构建语法分析器，请参阅12.2节中“关键字作为标识符”部分。

相比语法分析器，词法分析器需要动作的情况较少，不过在诸如需要修改词法符号类型或者文本的特定场景下，它们仍然相当有用。除了在对输入文本进行词法分析时执行动作之外，另一种修改词法符号本身的方法是查看词法分析后的词法符号流。

在本章中，我们学习了使用动作在语法中嵌入程序逻辑代码的方法，这些动作可以位于规则内，也可以位于规则外，通过**header**和**members**发挥作用。我们也看到了如何定义和引用规则的参数和返回值。随后，我们还使用了词法符号的属性，如**text**和**type**。合在一起，这些与动作相关的特性使我们能够自定义ANTLR生成的代码。

再次提醒，应尽可能地避免使用语法中的动作，因为它将一份语法绑定到了特定的目标编程语言上。不仅如此，动作还将语法绑定到了一个特定的程序上。不过，你可能并不在乎这件事情，因为你的公司一直以来使用的都是同一门语言，你的语法也只适用于特定的程序。在这样的情况下，基于简便或者效率（省去了建立语法分析树的开销）方面的原因，在语法中直接嵌入动作就变得非常有意义了。最重要的是，一些语法分析问题需要运行时的测试才能正确识别输入文本。在

下一章中，我们将会研究一种名为语义判定的任意布尔表达式，它可以动态地开启或者关闭某些备选分支。

第11章 使用语义判定修改语法分析过程

在上一章中，我们学习了如何在语法中嵌入动作，以便在语法分析的过程中执行应用的相关逻辑。无论如何，这些动作代码都不会影响语法分析器的语法分析过程，就好像记录日志的语句不会影响外围程序一样。我们的内嵌动作仅仅是计算一些值或者打印结果。但是，在一些罕见情况下，使用内嵌动作来修改语法分析过程是正确识别某些编程语言语句的唯一方法。在本章中，我们将会学习一种特殊的动作 `{...}?`，称为语义判定，它允许我们在运行时选择性地关闭部分语法。判定本身是布尔表达式，它会减少语法分析器的在语法分析过程中可选项的数量。一个令人难以置信的事实是，适当地减少可选项的数量会增强语法分析器的性能！

语义判定可以在两种常见的情况下发挥作用。第一，我们可能需要语法分析器处理同一门编程语言稍有差异的多个版本（方言）。例如，数据库供应商的SQL语法会随着时间演进。为了为这样的供应商编写数据库的前端模块，我们需要支持同一种SQL语言的不同版本。与之相似，Gnu的C编译器——gcc——需要处理ANSI C和自身提供的方言，这些方言提供了一些扩展，例如很好的“动态goto”特性。语义判定

允许我们通过命令行参数或者其他动态机制，在运行时选择所使用的方言。

第二个应用场景包括处理语法的歧义性（已在2.2中讨论过）。在某些编程语言中，相同的语法结构具有不同的含义，此时判定机制提供了一种方法，让我们能够在对相同输入文本的不同解释中做出选择。例如，在古老的Fortran语言中，`f(i)` 既可以是数组引用，也可以是函数调用，取决于`f`的定义是什么。这种情况下，两种语义的语法是相同的。编译器必须在符号表中查找该标识符，才能对输入作出正确解释。

语义判定给我们提供了这样一种途径：我们能够基于符号表来“关闭”对输入文本作出的错误解释。这使得语法分析器别无选择，只能采用正确的解释。

我们将通过Java和C++中的一些例子学习语义判定。随后，我们会深入研究它的细节，你也可以阅读15.7参考章节，其中包含了对细节的讨论。掌握了内嵌动作和语义判定，我们就能够胸有成竹地处理下一章的语言识别难题了。

11.1 识别编程语言的多种方言

首先，我们会学习如何使用语义判定来关闭Java语法中的一部分。通过在运行过程中对布尔表达式求值，它能够达到识别不同方言的目的。

在本例中，我们会看到，如何使同一个语法分析器在支持和不支持枚举类型之间自由切换。

Java语言在近年来的扩展加入了一些新的结构，例如，在Java 5之前，下列声明是非法的：

```
predicates/Temp.java
enum Temp { HOT, COLD }
```

与其编写两个编译器来处理这两种稍有差异的方言，Java自带的编译器javac有一个-source选项。下面的结果显示了我们试图以Java 1.4的版本编译enum的结果：

```
$ javac -source 1.4 Temp.java
Temp.java:1: enums are not supported in -source 1.4
(use -source 5 or higher to enable enums)
enum Temp { HOT, COLD }
^
1 error
$ javac Temp.java # javac 默认使用最新版本的方言，因此编译得以通过
```

引入枚举类型使得enum从一个标识符变成了一个关键字，引发了向后兼容问题。许多遗留代码将enum用作变量名，例如“int enum;”。如果能通过一个编译器选项来识别这样的早期方言，我们就无须仅仅为了编译通过而修改它们。

为初步了解javac处理多种方言的手段，我们将会编写一份语法来识别Java的部分片段：enum定义和赋值语句。我们的最终目标是编写出能

够正确识别Java 5之前和之后的版本的语法，不过，我们不会处理二者混杂的情况：`enum`同时被用作关键字和标识符是非法的。

```
enum enum { HOT, COLD }    // 在 Java 的任何版本中都是非法的
```

我们先简单了解上述Java子集语法的核心内容，然后再设法处理`enum`关键字。

```
predicates/Enum.g4
grammar Enum;
@parser::members {public static boolean java5;}

prog:  (  stat

        |  enumDecl
      )+
      ;

stat:  id '=' expr ';' {System.out.println($id.text+"="+$expr.text);} ;

expr
:  id
|  INT
;

```

到这里我们已经很熟悉上面的语法结构和动作了，所以下面进一步讨论`enum`的声明。

```
enumDecl
:  'enum' name=id '{' id (',' id)* '}'
    {System.out.println("enum "+$name.text);}
;

```

这条规则识别的句法是（简化的）枚举类型定义，但是其中并没有指出`enum`在某些时候是非法的。这就是关键所在：使用语义判定开启和关闭备选分支。

```
predicates/Enum.g4
enumDecl
:   {java5}? 'enum' name=id '{' id (',' id)* '}'
    {System.out.println("enum "+$name.text);}
;
```

`{java5}?` 判定在运行的时候求值，当结果为假时，该备选分支就被关闭。

你可能注意到了，我们使用规则`id`代替了常见的`ID`。这是由于`enum`在Java 5之前是合法的标识符（词法分析器将`enum`处理为关键字，而非标识符）。因此，我们就需要一条带语义判定的文法规则来表述这种选择。

```
predicates/Enum.g4
id  :   ID
     |   {!java5}? 'enum'
     ;
```

`{! java5}?` 判定允许`enum`在非Java 5模式下作为普通标识符使用。从字面上看，当`java5`为真时，它关闭了第二个备选分支。在内部，ANTLR语法分析器将规则`id`看作一个图数据结构，类似图11-1。

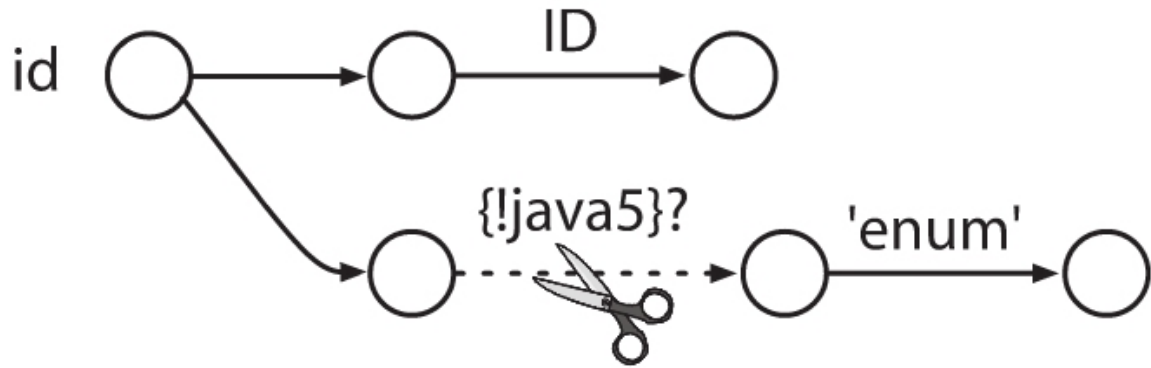


图11-1 将规则id看作一个图数据结构

图11-1中的剪刀表示，语法分析器在！java5为假（即java5为真）时剪掉了该分支。注意，由于判定结果是互斥的，enum声明和enum标识符也是互斥的。

我们可以使用grun来测试上述语法，不过我们首先需要一个可以识别方言切换的测试类。下面的TestEnum类支持使用一个-java5参数来开启Java 5模式。

```
predicates/TestEnum.java
int i = 0;
EnumParser.java5 = false; // 默认是非 Java5 模式
if ( args.length>0 && args[i].equals("-java5") ) {
    EnumParser.java5 = true;
    i++;
}
```

现在我们可以开始构建和编译了。

```
$ antlr4 -no-listener Enum.g4
$ javac Enum*.java TestEnum.java
```

让我们首先测试Java 5之前的模式，确保这种情况下，它允许enum作为合法的标识符，且不允许枚举类型。

```
⇒ $ java TestEnum
⇒ enum = 0;
⇒ EOF
  < enum=0

⇒ $ java TestEnum
⇒ enum Temp { HOT, COLD }
⇒ EOF
  < line 1:0 no viable alternative at input 'enum'
```

相比之下，Java 5模式不应该将enum看作标识符，但是应该允许枚举类型。

```
⇒ $ java TestEnum -java5
⇒ enum = 0;
⇒ EOF
  < line 1:0 no viable alternative at input 'enum'

⇒ $ java TestEnum -java5
⇒ enum Temp { HOT, COLD }
⇒ EOF
  < enum Temp
```

一切正常，在继续学习之前，让我们来看一看判定可被放置的位置。判定可以开启和关闭任何在通过判定后能被匹配的规则。这意味着，我们无须将{java5}? 判定放置在enumDecl中，可以把它提到该规则调用之前。

```
prog:  ( {java5}? enumDecl
        |  stat
      )+
      ;
```

它们在功能上是等价的，在本例中，两种放置方式仅仅存在风格上的差异。关键在于，语法分析器在抵达`enumDecl`中的`enum`之前，必定会在 `(...)+` 的第一个备选分支中的某个地方遇到一个判定。

这就是利用运行期开关支持多方言语法的编写方法。如果你希望编写一份真实的Java语法，你就需要在其中的相应规则中，集成这些我们称之为`enumDecl`和`id`的判定。

除此之外，在词法分析器中，语义判定也可以通过内嵌动作来发挥作用。

11.2 关闭词法符号

在本节中，我们将会重新解决上一章的问题，不过这次是通过在词法分析器而非语法分析器中使用判定。我们的主要思想是，令词法分析器中的判定动态地开启和关闭词法符号（`token`），而非语言中的词组（`phrase`）。我们会在Java 5之前的模式中，关闭把`enum`当作关键字的词法规则，将其作为一个常规的标识符处理。在Java 5模式中，我们将`enum`当作一个关键字类型的词法符号处理。这大大简化了语法分析

器，因为它可以通过常规的ID词法符号来匹配标识符，而无须使用id规则。

```
predicates/Enum2.g4
```

```
stat: ID '=' expr ';' {System.out.println($ID.text+"="+$expr.text);} ;

expr: ID
     | INT
     ;
```

词法分析器应当只在当前方言允许的情况下输出ID。为此，我们需要在匹配enum的词法规则中加入一个判定。

```
predicates/Enum2.g4
```

```
ENUM: 'enum' {java5}? ; // 必须放置在 ID 规则之前
ID : [a-zA-Z]+ ;
```

需要注意的是，判定出现在词法规则的右侧，而非像文法规则一样的左侧。这是由于在语法分析中，语法分析器会对之后的内容进行预测，因此需要在匹配备选分支之前进行判定。

而词法分析器不进行备选分支的预测。它们仅仅寻找最长的匹配文本，然后在发现整个词法符号后作出决策（我们将在参考章节中，尤其是15.7节中，进行深入学习）。

当java5为假时，该判定关闭了ENUM规则。当它为真时，ENUM和ID同时匹配了字符序列e-n-u-m，此时该输入存在歧义。ANTLR总是通过选择位置靠前的规则来解决词法歧义问题，也就是这里的ENUM。如

果我们把二者的位置反过来，那么无论**ENUM**是否被开启，词法分析器总是会将**e-n-u-m**匹配为**ID**。

这种词法判定的解决方案的优雅之处在于，我们无须在语法分析器中放置一个判定，用于在非Java 5模式下关闭**enum**结构。

```
predicates/Enum2.g4
// 这里无需判定，因为 'enum' 词法符号在 !java5 的情况下并未定义
enumDecl
:   'enum' name=ID '{' ID (',' ID)* '}'
    {System.out.println("enum "+$name.text);}
;
```

其中，备选分支开头的词法符号'**enum**'会寻找这样一个关键字词法符号。词法分析器只有在Java 5模式下才会将其输送给语法分析器，因此当**java5**为假时，**enumDecl**永远不会得到匹配。

现在，让我们确认一下基于词法分析器的解决方案能够正确处理这两种方言。首先，在非Java 5模式中，**enum**应当是一个标识符。

```
⇒ $ antlr4 -no-listener Enum2.g4
⇒ $ javac Enum2*.java TestEnum2.java
⇒ $ java TestEnum2
⇒ enum = 0;
⇒ EOf
  < enum=0
```

（注：该文件位于

<https://media.pragprog.com/titles/tpantlr2/code/predicates/TestEnum2.java>

。——译者注）

其次，由于`enum`是一个标识符，而非关键字，所以语法分析器绝不会去尝试匹配`enumDecl`。它别无选择，只能将`enum Temp{HOT, COLD}`当作赋值语句处理，从而发生了语法错误。

```
⇒ $ java TestEnum2
⇒ enum Temp { HOT, COLD }
⇒ EOF
< line 1:5 missing '=' at 'Temp'
  line 1:15 mismatched input ',' expecting '='
  line 1:22 mismatched input '}' expecting '='
```

在这里，ANTLR的错误恢复机制在开始匹配赋值语句时，意识到它缺少后半部分，于是开始丢弃词法符号，直至遇到下一个赋值语句为止。

在Java 5模式中，赋值给`enum`是非法的，但是枚举类型是合法的。

```
⇒ $ java TestEnum2 -java5
⇒ enum = 0;
⇒ EOF
< line 1:5 mismatched input '=' expecting ID
⇒ $ java TestEnum2 -java5
⇒ enum Temp { HOT, COLD }
⇒ EOF
< enum Temp
```

判定会拖慢词法分析器，如果我们希望完全避免它，我们可以去掉`ENUM`规则，然后将`enum`作为关键字处理。然后我们就可以按照10.3节中所做的那样，对词法符号的类型进行相应修改。

```
ID : [a-zA-Z]+  
    {if (java5 && getText().equals("enum")) setType(Enum2Parser.ENUM);}  
;
```

如果使用这种方式，我们就需要额外定义一个**ENUM**类型的词法符号。

```
tokens { ENUM }
```

出于效率和可读性的原因，应该尽量避免语法分析器中的内嵌判定。作为替代方案，我推荐使用本节中基于词法分析器的解决方案来处理Java中的enum问题。不过，需要注意的是判定也会同样拖慢词法分析器，所以最好完全不使用它们。

这就是语法分析器和词法分析器中，语义判定的基本语法和用法。语义判定为选择性的关闭部分语法提供了一种直接的方式，它允许我们使用同一份语法来识别相同语言的不同方言。此外，我们可以通过修改布尔表达式的值来在不同方言中动态地切换。现在，让我们研究一下第二种主要应用场景：在语法分析器中使用判定来解决输入文本的歧义问题。

11.3 识别歧义性文本

在上文中，我们了解了如何基于一个简单的布尔变量来关闭部分语法。这种情况并非用不同方式匹配相同的输入，我们仅仅希望关闭特

定的语言结构。现在，我们的目标是，在处理具有歧义的输入文本时，强制语法分析器只留下一种解释方式，而将其余的解释方式全部关闭。使用迷宫进行类比，当我们可以用同一条通行口令经多条路径通过一个迷宫时，我们就称该迷宫和通行口令是具有歧义的。判定就像是迷宫岔路口上的，可以开关的、用于通行的门。

语言的歧义是一件糟糕的事情吗？

聪明的编程语言设计者会有意识地避免歧义性结构，因为这会使得代码难以阅读。例如，**Ruby**中的`f[0]`既是数组`f`的第一个元素的引用，又是取函数`f()`返回的数组结果的第一个元素。更有意思的是，中间带空格的`f[0]`是将一个仅有一个元素`0`的数组当作参数传递给函数`f()`。上述情况发生的原因是在**Ruby**中，函数调用的括号是可选的。**Ruby**爱好者们现在推荐使用括号，因为它们的歧义实在太严重了。

在开始本节的学习之前，我必须指出，如果一份语法能够以多种方式匹配输入的文本，那么，通常情况下，这份语法是有问题的（a **grammar bug**）。在绝大多数语言中，语法本身仅仅说明如何解释所有有效语句（参见如上“语言的歧义是一件糟糕的事情吗？”栏目）。这意味着我们的语法对于每个输入的字符流，应当仅以一种方式进行匹配。如果有多种解释方式，我们就应该重写该语法，除去无效的解釋方式。

即便如此，在某些编程语言中，仍然存在一些仅靠语法本身不足以区分其含义的语句。这些语言的语法需要具有一定的歧义性，不过，歧义性语句在具体的上下文中就会具有清晰的含义，例如标识符的定义（作为类名还是函数名）。我们需要利用判定询问当前的上下文，以对歧义性文本作出正确的解释。如果一个判定能够成功地解决一种输入文本的语法歧义问题，我们就称这种输入是上下文相关的（**context-sensitive**）。

在本节中，我们计划研究一些C++的细节。据我所知，C++是最难进行准确语法分析的编程语言。我们将会首先学习如何区分函数调用和构造器风格的类型转换，然后学习如何区分声明和表达式。

1. 正确识别C++中的T (0)

在C++中，表达式T (0) 既是函数调用，又是构造器风格的类型转换，它的准确含义取决于T是函数名还是类型名。因为相同的语句能够用两种方式解释，所以该表达式具有歧义性。为进行正确的解释，语法分析器需要依据T的定义关闭某个备选分支。下面的简化版的C++表达式规则包含两个判定，能够检查ID是函数名还是类型名。

```
predicates/CppExpr.g4
```

```
/** 前两个备选分支中使用了理想化的判定作为区分这两种情况的 Demo */
expr:  {<<isfunc(ID)>>}? ID '(' expr ')' // 一个参数的函数调用
      | {<<istype(ID)>>}? ID '(' expr ')' // 构造器风格的对 expr 的转换
      | INT                               // 整数常量
      | ID                                // 标识符
      ;
```

expr规则的可视化展示如图11-2所示，在前两个备选分支之前存在切断点（cut point）。

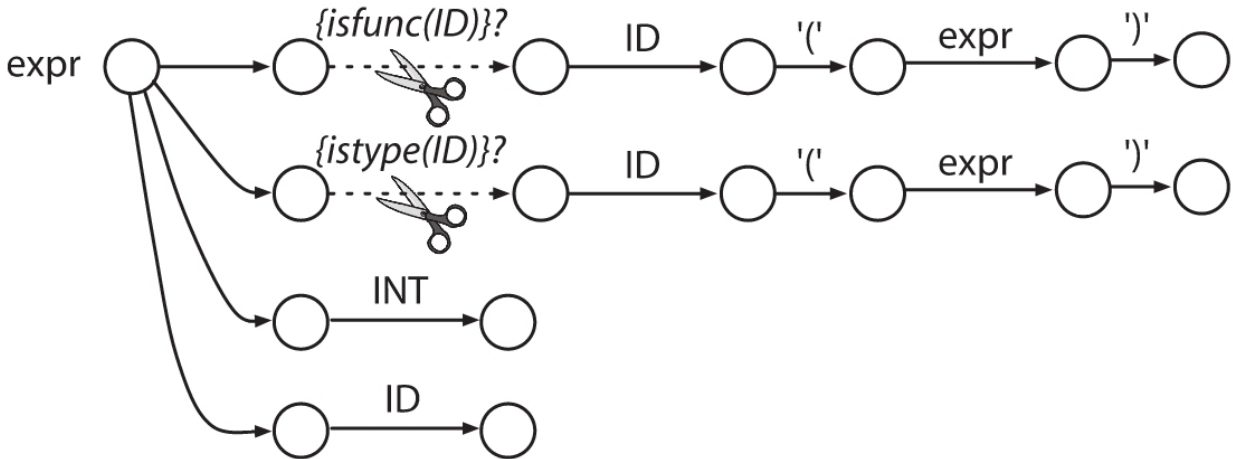


图11-2 expr规则的可视化展示

你可能会有疑问，为什么我们不将这两个备选分支组合成一条规则，来处理两种情况（函数调用和类型转换）呢？原因之一是这样做会使语法分析树遍历器的工作变得复杂。如果不分配两个方法的话，就会存在一个enterCallOrTypecast（）方法。在此方法中，我们必须手动区分这两种情况，但这还不是最糟的。

更大的问题在于，有歧义的备选分支很少像这个例子一样结构相同。例如，函数调用备选分支也需要处理没有参数的情况，如T（）。它可不是一个合法的类型转换，因此，将两个备选分支组合在一起实际上是行不通的。同样，有歧义的备选分支可能相距甚远，这就是我们接下来要讨论的话题。

2. 正确识别C++中的T (i)

考虑到上文讨论的表达式的变体T (i)。简单起见，我们先假设我们的C++子集中不存在构造器风格类型转换。此时，T (i)就一定是一个函数调用了。不幸的是，按照语法，它也是一个合法的声明。它等价于T i，定义了一个T类型的变量i。区分二者的唯一方法仍然是通过上下文。如果T是类型名，那么T (i)就是变量i的声明。否则，它就是将i作为参数的函数调用。

我们可以通过一个小型的C++语法来展示这些位于不同规则中的、有歧义的备选分支。我们不妨假设C++的语句只包含声明和表达式。

predicates/CppStat.g4

```
stat:  decl ';' {System.out.println("decl "+$decl.text);}
      |  expr ';' {System.out.println("expr "+$expr.text);}
      ; 前两个备选分支中使用了理想化的判定作为区分这两种情况的 Demo
```

一个声明既可以是T i，也可以是T (i)。

predicates/CppStat.g4

```
decl:  ID ID          // 例如 "Point p"
      |  ID '(' ID ')' // 例如 "Point (p)", 和 ID ID 等价
      ;
```

假设一个表达式只能是整数常量、简单标识符或者单参数的函数调用。

```

predicates/CppStat.g4
expr:    INT           // 整数常量
      |    ID           // 标识符
      |    ID '(' expr ')' // 函数调用
      ;

```

如果我们用上述语法测试输入“f (i) ; ”，我们会得到语法分析器给出的歧义性警告（使用了-diagnositics选项）。

```

⇒ $ antlr4 CppStat.g4
⇒ $ javac CppStat*.java
⇒ $ grun CppStat stat -diagnostics
⇒ f(i);
⇒ Eof
< line 1:4 reportAttemptingFullContext d=0, input='f(i);'
  line 1:4 reportAmbiguity d=0: ambigAlts={1, 2}, input='f(i);'
  decl f(i)

```

语法分析器首先告诉我们，它在试图使用简单的SLL (*) 策略对输入进行语法分析时发现了一个问题。由于该策略失败了，语法分析器就换用了更加强大的ALL (*) 机制。详见13.7节。使用了这种全语法分析算法（full grammar analysis algorithm）后，语法分析器再次发现了问题。此时，它知道输入确实存在歧义。如果语法分析器没有发现问题，它就会打印reportContextSensitivity消息。我们稍后将会学习有关歧义性警告的更多知识。

输入文本同时匹配了decl的第二个备选分支和expr的第三个备选分支。语法分析器必须在stat规则中做出选择。给定两个可用的备选分支，语

法分析器解决歧义问题的策略是选择靠前的那一个（decl）。这就是语法分析器将“f (i) ; ”解释成声明而非函数调用的原因。

假如我们拥有一个能够告诉我们某个标识符是不是类型名的“神谕”，我们就能通过在相应的备选分支之前放置判定来解决该歧义问题。

```
predicates/PredCppStat.g4
decl:  ID ID                                // 如 "Point p"
      | {istype()}? ID '(' ID ')'          // 如 "Point (p)", 即 ID ID
      ;

expr:  INT                                  // 整数
      | ID                                  // 标识符
      | {!istype()}? ID '(' expr ')'       // 函数调用
      ;
```

判定中的istype（）辅助方法从语法分析器处获取当前词法符号中的文本，然后在我们预定义的类型表中查询该文本。

```
predicates/PredCppStat.g4
@parser::members {
Set<String> types = new HashSet<String>() {{add("T");}};
boolean istype() { return types.contains(getCurrentToken().getText()); }
}
```

当我们使用这份带判定的语法再次进行测试时，输入“f (i) ; ”被正确地解释成了函数调用表达式，而非声明。输入“T (i) ; ”也被正确解释成了声明。

```
⇒ $ antlr4 PredCppStat.g4
⇒ $ javac PredCppStat*.java
⇒ $ grun PredCppStat stat -diagnostics
⇒ f(i);
⇒ E0f
  < expr f(i)
⇒ $ grun PredCppStat stat -diagnostics
⇒ T(i);
⇒ E0f
  < decl T(i)
```

图11-3所示的语法分析树（使用`grun-ps file.ps`命令创建）清楚地显示出，语法分析器正确地解释了输入的文本。

语法分析树中的关键节点是T和f的带下划线的父节点。这些内部节点告诉我们语法分析器匹配到的类型。记住，语言识别的根本意义在于区分不同语句的差异，识别出语言各组成部分。我们当然可以使用“.”（匹配一个或多个任意字符）来匹配所有可能的输入文本，但是它不能给我们传递任何信息。语言类应用程序的关键在于从输入中获取正确的语言结构。

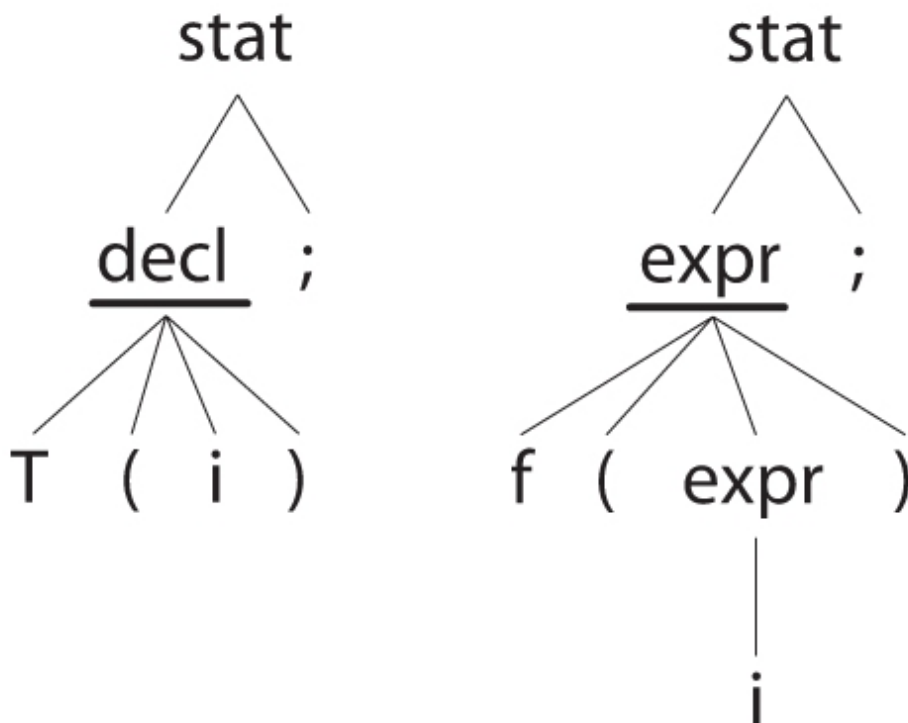


图11-3 正确解释了输入文本的语法分析树示例

因为判定除去了错误的解释方式，上述C++例子中的歧义性消失了。不幸的是，还存在一些判定无法解决的歧义问题。下面让我们再研究一个C++的例子，看看这种情况是如何发生的。

3.正确识别C++中的T (i) [5]

C++引人入胜的地方在于，它的一些语句具有两种有效含义。考虑C++中的T (i) [5]。即使我们知道T是一个类型名，上述语句在语法上也能够同时解释成声明和表达式。这意味着，我们无法通过测试标识符T来切换解释方式，因为当T是类型名时就存在两种解释方式。

解释为声明的方式是将它当作一个具有5个T类型元素的数组：`T i[5]`。

解释为表达式的方式是将*i*转换为类型T，然后对其进行索引操作。

C++语言规范解决这种歧义问题的方案是：总是选择声明而非表达式。它明白无误地告诉人类如何解释`T (i) [5]`，但是，即便我们加入了语义判定，编写一份通常意义上的无歧义的语法也是不可能的。

幸运的是，语法分析器自动地解决了该歧义问题，所以它能够正常工作。语法分析器解决歧义问题的原则是选择位置靠前的备选分支。因此，我们只需要确保`stat`中，`decl`备选分支放在`expr`备选分支之前即可。

最后，在对C++进行语法分析时，还有一种复杂情况需要考虑。

4.解决前向引用问题

一个真实的C++语法分析器在语法分析的过程中遇到各种名字时，必须将它们记录下来，以便计算出之前章节提到的类型表或者其他用于区分函数名和类型名的表。在C++中，记录符号的过程稍有投机取巧之嫌，不过原则上，它不算是一个问题。在之前章节构建计算器的例子中，我们已经学过用键值对记录变量的方法。问题在于，有些时候，C++允许对符号的前向引用，例如方法和变量名。这意味着我们可能在遇到`T (i)`的时候还不知道T是不是一个函数名。

你可能开始明白了，对C++进行语法分析为什么这么难。唯一的解决方案是对输入文本或者输入文本的内部表示（如语法分析树）进行多趟扫描。

使用ANTLR，最简单的方法是将输入文本处理成词法符号流，然后快速扫描它并记录所有的符号定义，然后再“凭感觉”对这些词法符号进行语法分析，获得正确的语法分析树。

虽然绝大多数编程语言不会遇到这样的噩梦般的歧义问题，但是由于算术运算符的存在，几乎每种编程语言都具有歧义性。例如，在5.4节中，我们看到了 $1+2*3$ 的歧义性：我们可以将它解释成 $(1+2)*3$ 或者 $1+(2*3)$ 。

如果我们将语义判定看作能够开启和关闭备选分支的简单布尔表达式，它的行为就显得十分简单了。不幸的是，包含多重判定和内嵌动作的语法会变得非常复杂。本书中的参考章节详细介绍了ANTLR中判定的使用时机和方法。如果你不打算在语法中大量使用混杂了动作的判定，你可以跳过15.7节。在后续章节的学习中，这些细节将会有助于解释一些令人困惑的语法问题。

掌握通过动作和语义判定来定制语法分析器的方法后，我们就拥有了强大的武器，在下一章中，我们计划利用在本书第三部分中学到的知识，解决一些非常难的识别问题。

第12章 掌握词法分析的“黑魔法”

在本书的第二部分中，我们已经学到了一些高级技巧。我们知道了如何在语法分析的过程中执行任意代码，并且学会了使用语义判定来修改识别语法的过程。现在是时候将它们付诸实践，解决一些充满挑战的语言识别难题了。这次，让我们从词法分析器入手，而非语法分析器。

根据我的经验，如果一个语言识别问题难以解决，那么大多数情况下，最棘手的部分位于词法分析器中（当然，C++是一个例外，它到处的都很棘手）。这和我们的直觉相悖，因为迄今为止我们看到的词法规则都十分简单，例如标识符、整数和算术运算符。不过，考虑一个看上去非常简单的二字符序列：>>。Java词法分析器可以将它匹配成右移运算符或者两个>符号，后者出现在泛型声明的结尾处，例如 `List<List<String>>>`。

最根本的问题在于，词法分析器进行词法分析工作，有时也需要一些上下文信息对词法符号作出决策，但是这些上下文信息只被语法分析器持有。我们将在12.2中探讨这个问题。在讨论过程中，我们还会看到“关键字同时也能作为标识符”的问题，并尝试构建一个用于处理Python的上下文相关的换行符的词法分析器。

随后我们会看到的问题包括孤岛语言（island language）——一种语句中包含“孤岛部分”和“海洋部分”的语言，它的孤岛部分是我们所需处理的，海洋部分是我们不关心的内容。这样的例子包括XML和类似StringTemplate的模板语言。为了完成这种语言的语法分析，我们需要孤岛语法和词法模式，它们将在12.3节中介绍。

最后，我们会根据XML规范，用ANTLR编写一个XML词法分析器和语法分析器。它是一个很好的例子，展示了如何处理包含不同内容（区域）的输入文本、如何划分语法分析器和词法分析器的界线，以及如何处理非ASCII字符。

作为热身，我们首先学习忽略但是不丢弃特定输入区域——例如注释和空白字符——的方法。这项技术可用于解决一些语言翻译问题，我们会展示最常见的例子。

12.1 将词法符号送入不同通道

绝大多数编程语言忽略词法符号间的空格和注释，这意味着它们可以出现在任何地方。这就给语法分析器带来了一个难题，它必须时刻考虑两种可选的词法符号的存在：空白字符和注释。常见的解决方案是，令词法分析器匹配这些词法符号并丢弃，这就是我们在本书中所做的。例如，6.4节的Cymbol语法使用词法分析器指令skip丢弃空白字符和注释。

```
examples/Cymbol.g4
```

```
WS : [ \t\n\r]+ -> skip ;
```

```
SL_COMMENT
```

```
: '//' .*? '\n' -> skip
```

```
;
```

在绝大多数情况下，因为注释不影响生成的代码，这种方案表现出色——例如编译器。另一方面，如果我们在编写一个将遗留代码翻译成现代编程语言的翻译器，那就真的需要保留其中的注释了，因为它们是代码的一部分。这带来一个难题：我们希望保留注释和空白字符，但是不希望在语法分析器中时常检查它们，这会加重语法分析器的负担。

1. 填充词法符号通道

ANTLR的解决方案是将类似标识符的正常词法符号送入语法分析器对应的通道，其余内容送入另外一个通道。通道就像不同的广播频率。

词法规则负责将词法符号放入不同的通道，**CommonTokenStream**类负责只对语法分析器暴露其中一个通道。在此期间，

CommonTokenStream保留了原先词法符号的顺序，这样我们就能获取特定的语言词法符号前后处的注释内容。图12-1从**CommonTokenStream**的角度展示了一个C语言词法分析器将注释和空白字符放入隐藏通道的过程。

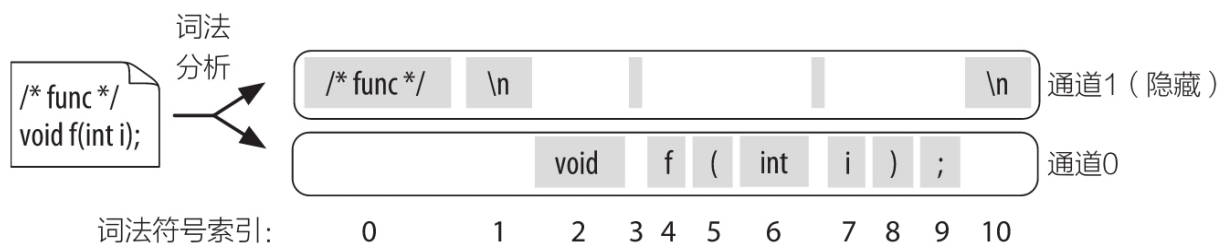


图12-1 注释和空白字符放入隐藏通道的过程

我们可以轻易地将注释和空白字符分别送入不同的通道，而正常的词法符号仍然位于默认的0通道。

这样，我们就能分别获取注释和空白字符了。

通过在对应的词法规则上放置词法分析器指令**channel (...)**，我们可以将词法符号送入不同通道。让我们试着使用这种方法修改Cymbol语法，将注释放入隐藏通道2，空白字符放入隐藏通道1，如图12-2所示。

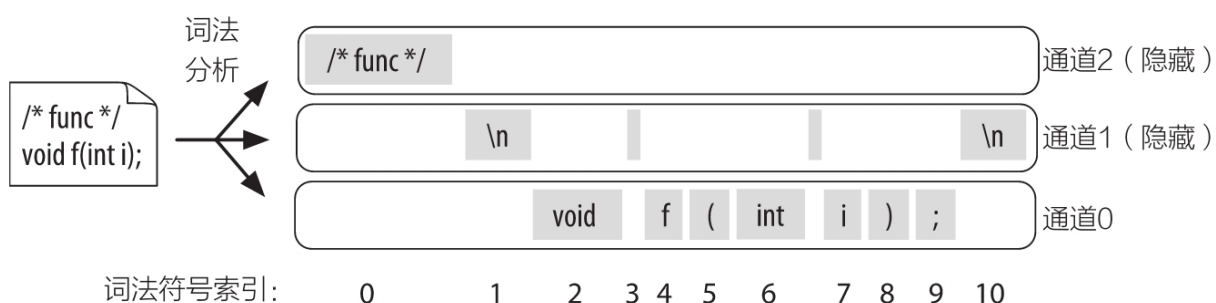


图12-2 将词法符号送入不同通道

```
lexmagic/Cymbol.g4
```

```
WS : [ \t\n\r]+ -> channel(WHITESPACE) ; // channel(1)
```

```
SL_COMMENT
```

```
: '//' .*? '\n' -> channel(COMMENTS) // channel(2)
;
```

常量WHITESPACE和COMMENTS源于语法中的声明。

```
lexmagic/Cymbol.g4
```

```
@lexer::members {
```

```
    public static final int WHITESPACE = 1;
```

```
    public static final int COMMENTS = 2;
```

```
}
```

ANTLR将channel（HIDDEN）翻译为Java代码_channel=HIDDEN，它将Lexer类的成员_channel设置为常量HIDDEN。我们可以使用任何合法的Java标识符作为channel（）指令的参数。

利用grun进行的测试结果显示，注释出现在了通道2上，空白字符出现在了通道1上，其他的词法符号仍旧出现在了默认通道上。

```
⇒ $ antlr4 Cymbol.g4
```

```
⇒ $ javac Cymbol*.java
```

```
⇒ $ grun Cymbol file -tokens -tree
```

```
⇒ int i = 3; // testing
```

```
⇒ EOf
```

```
⚡ [@0,0:2='int',<10>,1:0]
```

```
  [@1,3:3=' ',<24>,channel=1,1:3]
```

```
<-- HIDDEN channel 1
```

```
  [@2,4:4='i',<22>,1:4]
```

```
  [@3,5:5=' ',<24>,channel=1,1:5]
```

```
<-- HIDDEN channel 1
```

```

[@4,6:6='<11>',1:6]
[@5,7:7=' ',<24>,channel=1,1:7]          <-- HIDDEN channel 1
[@6,8:8='3',<23>,1:8]
[@7,9:9=';',<13>,1:9]
[@8,10:10=' ',<24>,channel=1,1:10]         <-- HIDDEN channel 1
[@9,11:21='// testing\n',<25>,channel=2,1:11] <-- HIDDEN channel 2
[@10,22:21='<EOF>',<-1>,2:22]
(file (varDecl (type int) i = (expr 3) ;)) <-- parse tree

```

语法分析树也表现正常，这意味着语法分析器对输入作出了正确的解释。即，语法分析器并没有遇到任何注释。接下来，我们将会学习在语言类应用程序中访问隐藏通道注释的方法。

2.访问隐藏通道

为展示从语言类应用程序中访问隐藏通道的方法，让我们来编写一个语法分析树监听器，将声明之后的注释移到声明之前，并将它们改写为/*...*/风格。例如，对于下列输入：

```

lexmagic/t.cym
int n = 0; // 定义一个计数器
int i = 9;

```

我们预期的输出为：

```

/* 定义一个计数器 */
int n = 0;
int i = 9;

```

我们的基本策略是使用TokenStreamRewriter重写词法符号流，即在4.5节中“重写输入流”部分中所完成的工作。在发现变量定义时，我们的

程序会提取右侧的注释（如果有的话），然后将它插入到声明的第一个词法符号之前。下面是一个名为CommentShifter的Cymbol语法分析树监听器，它位于名为ShiftVarComments的测试类中：

```
lexmagic/ShiftVarComments.java
第1行 public static class CommentShifter extends CymbolBaseListener {
-     BufferedTokenStream tokens;
-     TokenStreamRewriter rewriter;
-     /** 创建一个绑定到词法符号流上的 TokenStreamRewriter
5         * 位于 Cymbol 词法分析器和语法分析器之间
-         */
-     public CommentShifter(BufferedTokenStream tokens) {
-         this.tokens = tokens;
-         rewriter = new TokenStreamRewriter(tokens);
10    }
-
-     @Override
-     public void exitVarDecl(CymbolParser.VarDeclContext ctx) {
-         Token semi = ctx.getStop();
15         int i = semi.getTokenIndex();
-         List<Token> cmtChannel =
-             tokens.getHiddenTokensToRight(i, CymbolLexer.COMMENTS);
-         if ( cmtChannel!=null ) {

-             Token cmt = cmtChannel.get(0);
20         if ( cmt!=null ) {
-             String txt = cmt.getText().substring(2);
-             String newCmt = "/* " + txt.trim() + " */\n";
-             rewriter.insertBefore(ctx.start, newCmt);
-             rewriter.replace(cmt, "\n");
25         }
-     }
- }
- }
```

所有的事情都发生在exitVarDecl（）中。首先，我们取得声明语句中的分号的词法符号索引值（第14行）因为我们会寻找它后面的注释。第17行询问词法符号流，在该分号右侧的COMMENT通道中是否存在隐

藏的词法符号。简单起见，这份代码假定每个声明后仅存在一条注释，因此第19行从列表中获取第一条注释。接下来，我们将旧的注释改写成新风格的注释，然后使用`TokenStreamRewriter`将它插入到变量声明之前（第23行）。最后，我们将原先的注释用换行符代替（第24行），相当于将它移除。

测试程序本身没有什么新意，唯一需要注意的是在它的末尾处，我们调用`TokenStream-Rewriter`类的`getText()`方法获得了重写后的输入。

```
lexmagic/ShiftVarComments.java
CymbolLexer lexer = new CymbolLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
CymbolParser parser = new CymbolParser(tokens);
RuleContext tree = parser.file();
ParseTreeWalker walker = new ParseTreeWalker();
➤ CommentShifter shifter = new CommentShifter(tokens);
➤ walker.walk(shifter, tree);
➤ System.out.print(shifter.rewriter.getText());
```

下面是构建和测试的步骤：

```
$ antlr4 Cymbol.g4
$ javac Cymbol*.java ShiftVarComments.java
$ java ShiftVarComments t.cym
/* define a counter */
int n = 0;
int i = 9;
```

需要注意的是，如果我们丢弃了空白字符，而非将它们送入隐藏通道，那么输出结果就会挤在一起，如“`intn=0;`”。通过将输入的词法符

号分类，词法符号通道解决了一个棘手的语言翻译问题。接下来，我们一起研究一个与词法符号本身的构造过程相关的问题。

12.2 上下文相关的词法问题

考虑这样一个句子“**Brown leaves in the fall**”。它是有歧义的，因为存在两种解释。如果我们指的是树木的叶子，这句话就是在描述一种自然现象。但是，如果我们正在谈论一位Jane Brown女士，这句话的意义就完全被上下文改变了。“**Leaves**”就从名词变成了动词。这种情况类似我们在11.3节中解决的问题：C++中存在上下文相关的语句，如**T (0)**既可以是函数调用语句，也可以是类型转换语句，它的具体含义取决于当前程序中的**T**定义。由于我们的C++词法分析器输送给语法分析器的是含义模糊的通用**ID**词法符号，上述歧义性的影响更加显著。因此，我们需要在文法规则中加入语义判定来选择不同的备选分支。

此外，我们也可以令词法分析器向语法分析器输送更精确的词法符号，例如由上下文决定的**FUNCTION_NAME**或**TYPE_NAME**（对于“**Brown leaves**”这样的输入，我们就需要令词法分析器送出词法符号序列**ADJECTIVE NOUN**和**PROPER_NAME VERB**）。不幸的是，这种做法只是将上下文相关的问题转移给词法分析器而已，而词法分析器能够得到的上下文信息不如语法分析器多。这就是我们在之前章节中的文法规则中加入判定，而非依据词法上下文向语法分析器输送更精确的词法符号的原因。

黑科技警报：语法分析器反馈

解决上述问题的常用方案包括从语法分析器向词法分析器发送反馈，这样词法分析器就可以向语法分析器输出更加精确的词法符号。由于判定数量的减少，语法分析器可以变得更加简单。然而，这种方案在ANTLR语法中是行不通的，因为ANTLR自动生成的语法分析器经常在词法符号流中进行非常远的前瞻以作出语法分析决策。这意味着，远在语法分析器能够执行提供上下文信息的行为之前，词法分析器就需要将字符流处理为词法符号。

很多情况下，我们不可避免地需要在对输入字符流进行词法分析的过程中处理上下文相关问题。在本节中，我们将会看到三个上下文相关的词法问题。

- 相同的字符序列在语法分析器中具有不同含义。我们将会处理广为人知的“关键字也能作为标识符”问题。

- 相同的字符序列可以是一个或者多个词法符号。我们会看到如何处理Java中的>>，它既是两个泛型的结束符，也是一个右移运算符。

- 相同的字符序列在某些情况下需要被忽略，某些情况下需要被语法分析器识别。我们将会学习如何区分Python的物理（physical）换行符和逻辑（logical）换行符。解决这个问题同时需要我们在前两章中学到的词法动作和语义判定技术。

1.关键字作为标识符

许多编程语言——无论新老——都允许关键字在某些上下文中作为标识符使用。在Fortran中，我们可以编写类似`end=goto+if/while`的代码。C#通过它的Language-Integrated Query (LINQ) 功能提供了对SQL的支持。SQL查询语句以关键字`from`开始，但是我们也可以把`from`当作变量使用：`x=from+where;`。这是一个没有歧义的表达式，而非一个查询，因此词法分析器不应该将该`from`当作标识符。问题在于，词法分析器并不会对输入文本进行语法分析，也就无从得知需要将哪种词法符号送给语法分析器，是`KEYWORD_FROM`，还是`ID`。

我们可以通过两种方式允许关键字在某些上下文中作为标识符。第一种是令词法分析器将所有的关键字当作关键字类型的词法符号送给语法分析器，然后编写一条文法规则`id`，该规则匹配`ID`和任意的关键字。第二种是令词法符号将所有的关键字当作标识符，然后我们在语法分析器中编写如下判定来对标识符的名字进行测试：

```
keyIF : {_input.LT(1).getText().equals("if")}? ID ;
```

在巴黎

我曾于20世纪80年代后期在巴黎工作，很快，在打电话时，我发现了一个问题。当接听电话的人问我是谁时，我会回答`Monsieur Parr`，但是

Parr的发音和part——动词“to leave”的第三人称单数形式——非常相似。所以我的回答听上去就像是我要挂电话了。真搞笑。

有一个有趣的法语的绕口令，其中的每个单词的实际意义都需要上下文信息才能得出：“Si six cent scies scient six cent saucisses, six cent six scies scieront six cent six saucissons.”它的书面形式虽然充满了重复单词，但是含义十分清楚。然而，当读出来时，这句话的英文发音就像是：“See see saw, see see see saw, sawcease, see saw see see seeron see saw see sawcease.”翻译过来就是：“If six hundred saws saw six hundred sausages, six hundred and six saws will saw six hundred six sausages.”（如果600把锯子锯600根香肠，那么606把锯子就锯606根香肠）不要嘲笑法语中的si、six、scies和scient发音几乎相同。在英语中甚至存在一些单词写法相同，但是发音不同的现象，例如read（现在时）和read（过去时）！

这种方式非常丑陋，还可能很慢，因此我坚持使用第一种方式（出于完整性的考虑，我在PredKeyword.g4中留下了一个能够对关键字作出有效判定的小例子）。

下面是一份示例语法，它展示了我所推荐的方式，能够匹配类似“if if then call call;”的诡异输入：

```
lexmagic/IDKeyword.g4
```

```
grammar IDKeyword;
```

```
prog: stat+ ;
```

```
stat: 'if' expr 'then' stat  
    | 'call' id ';' ;  
    | ';' ;
```

```
expr: id ;
```

```
id : 'if' | 'call' | 'then' | ID ;
```

```
ID : [a-z]+ ;
```

```
WS : [ \r\n]+ -> skip ;
```

简而言之，这种方式将所有对词法符号ID的引用替换成了对文法规则id的引用。如果你正在处理一种不同上下文对应不同关键字集合的编程语言，你就需要多个id备选分支（每个上下文一个）。

下面是IDKeyword语法的构建和测试步骤：

```
⇒ $ antlr4 IDKeyword.g4  
⇒ $ javac IDKeyword*.java  
⇒ $ grun IDKeyword prog  
⇒ if if then call call;  
⇒ E0F
```

如图12-3所示生成的语法分析树显示，第二个if和第二个call被当作了标识符处理。

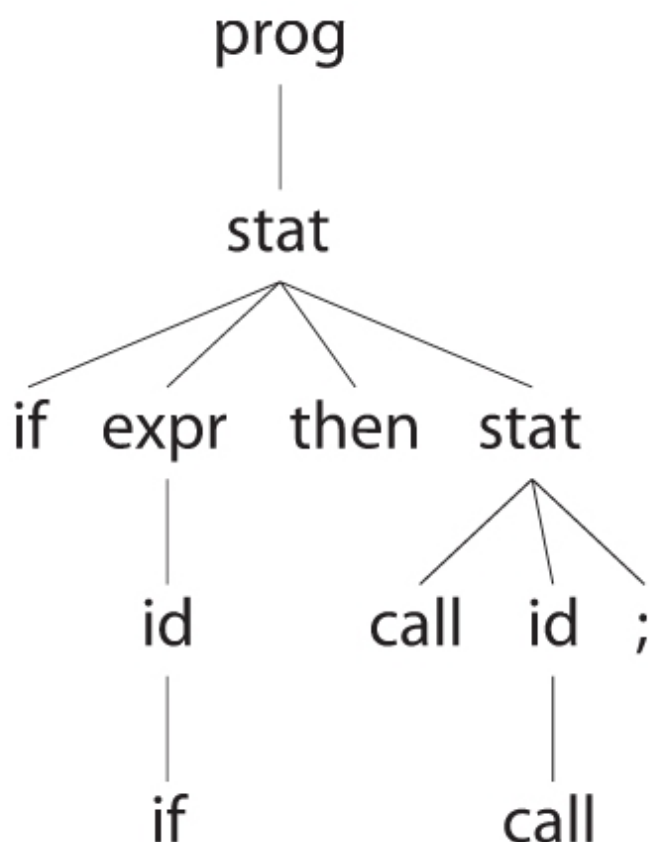


图12-3 第二个if和第二个call被当作标识符处理的语法分析树

在这个问题中，词法分析器必须决定输出关键字类型的词法符号还是标识符类型的词法符号，但是它无需关心这些词法符号究竟由哪些字符组成。接下来，我们将会解决一个问题：词法分析器不知道每个词法符号由多少个字符构成。

2.避免最长匹配带来的歧义性

通常，词法分析器生成器会作出这样的假设：在每个位置上，词法分析器应当尽可能地匹配最长的词法符号。基于该假设的词法分析器的

表现最为自然。例如，对于C语言中的+=，词法分析器应当匹配出单一的词法符号+=，而非两个词法符号+和=。然而，我们必须区分一些例外情况。

在C++中，我们不能连续放置嵌套的泛型尖括号，例如A<B<C>>。我们必须在最后两个尖括号之间加入一个空格：A<B<C>>，这样词法分析器才不会将两个尖括号误认为右移运算符>>。仅仅为了规避词法问题而让C++的设计者修改语言本身是不现实的。

有多种解决该问题的方案，最简单的一种是：令词法分析器从不将>>序列匹配为右移运算符，而将两个>符号输送给语法分析器，后者可以利用上下文信息对其进行适当的组装。例如，C++语法分析器中识别表达式的规则可以匹配两个右尖括号，而非单一的右移运算符。回顾一下4.3节中的Java语法，你会发现这种方案的一个范例。下面是expr规则中的两个备选分支，它们将单字符的词法符号组装成多字符的运算符：

```
tour/Java.g4
| expression ('<' '<' | '>' '>' '>' | '>' '>') expression
| expression ('<' '=' | '>' '=' | '>' | '<') expression
```

让我们看看当输入右移运算符时，词法分析器向语法分析器输送的词法符号。

```

⇒ $ antlr4 Java.g4
⇒ $ javac Java*.java
⇒ $ grun Java tokens -tokens
⇒ i = 1 >> 5;
⇒ E0F
  < [@0,0:0='i',<98>,1:0]
    [@1,1:1=' ',<100>,channel=1,1:1]
    [@2,2:2='=',<25>,1:2]
    [@3,3:3=' ',<100>,channel=1,1:3]
    [@4,4:4='1',<91>,1:4]
    [@5,5:5=' ',<100>,channel=1,1:5]
    [@6,6:6='>',<81>,1:6]          <-- 两个 '>' 词法符号, 而非单个 '>>'
    [@7,7:7='>',<81>,1:7]
    [@8,8:8=' ',<100>,channel=1,1:8]
    [@9,9:9='5',<91>,1:9]
    [@10,10:10=';',<77>,1:10]
    [@11,11:11='\n',<100>,channel=1,1:11]
    [@12,12:11='<EOF>',<-1>,2:12]

```

下面是输入嵌套泛型时的词法符号流:

```

⇒ $ grun Java tokens -tokens
⇒ List<List<String>> x;
⇒ E0F
  < [@0,0:3='List',<98>,1:0]
    [@1,4:4='<',<5>,1:4]
    [@2,5:8='List',<98>,1:5]
    [@3,9:9='<',<5>,1:9]
    [@4,10:15='String',<98>,1:10]
    [@5,16:16='>',<81>,1:16]
    [@6,17:17='>',<81>,1:17]
    [@7,18:18=' ',<100>,channel=1,1:18]
    [@8,19:19='x',<98>,1:19]
    [@9,20:20=';',<77>,1:20]
    [@10,21:21='\n',<100>,channel=1,1:21]
    [@11,22:21='<EOF>',<-1>,2:22]

```

下面我们会生成上述输入对应的语法分析树，它们清晰地显示出>词法符号的用法。

```

⇒ $ grun Java statement -gui
⇒ i = 1 >> 5;
⇒ E0F
⇒ $ grun Java localVariableDeclarationStatement -gui

```

```

⇒ List<List<String>> x;
⇒ E0F

```

如图12-4所示的两棵语法分析树的根节点分别是statement和localVariableDeclaration-Statement规则，其中尖括号已经被高亮标记。

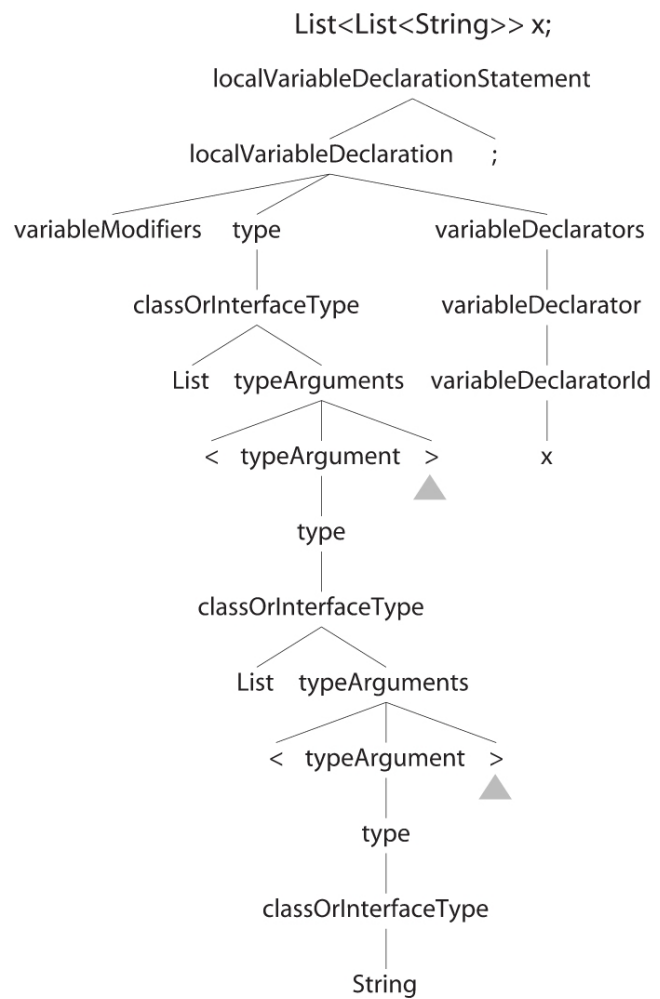
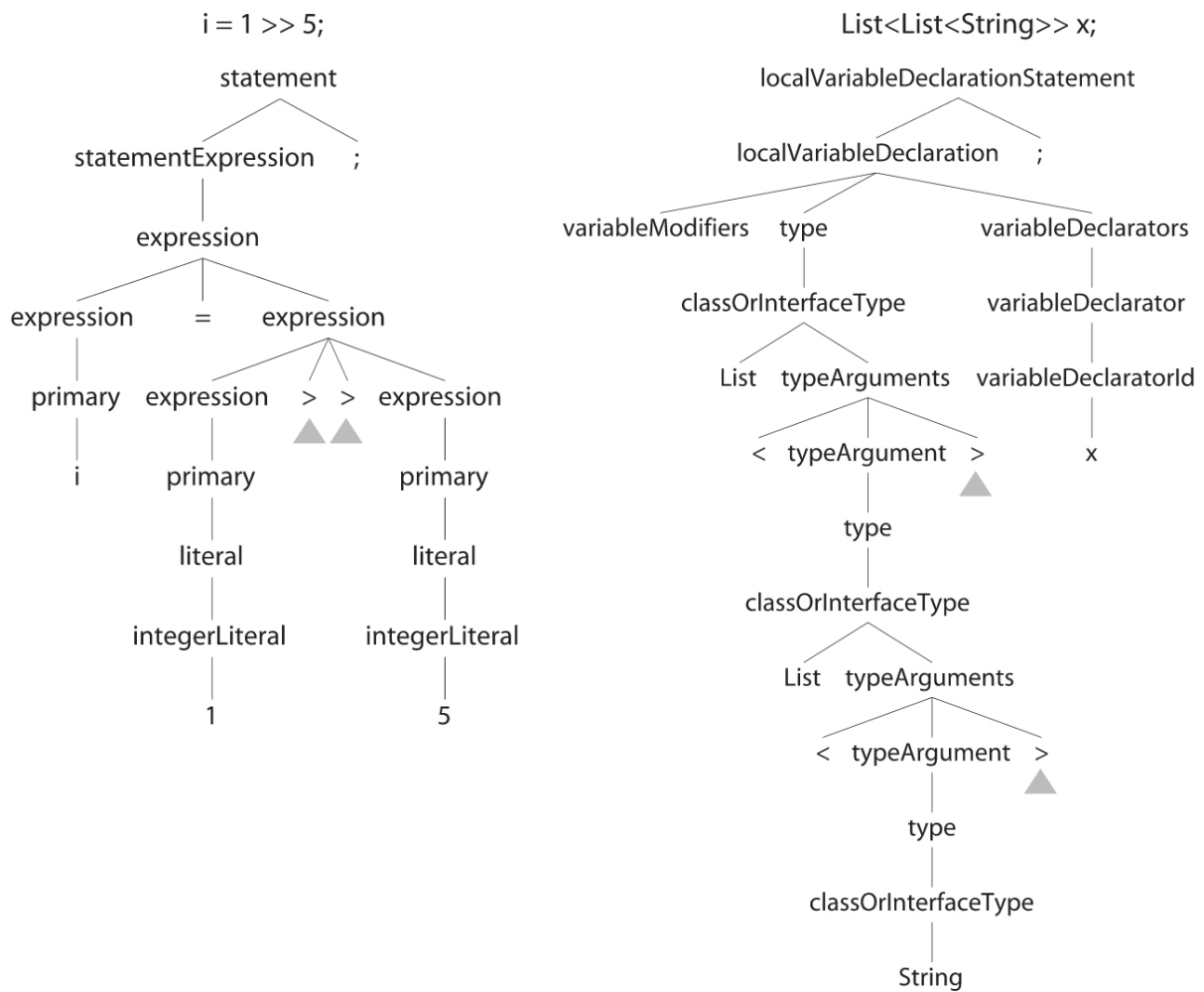


图12-4 尖括号被高亮标记的语法分析树

将右移运算符作为两个单独的右尖括号处理的唯一问题在于：语法分析器同样会接受中间包含空格的尖括号>>。欲解决这个问题，我们可以在语法中加入语义判定，或者使用监听器/访问器检查生成的语法分析树，确保多个>词法符号的列序号是相邻的。在语法分析的过程中使用判定的效率不高，所以最好在语法分析结束后检查右移运算符的正确性。毕竟，大多数语言类应用程序都需要对语法分析树进行遍历

（在表达式规则中使用判定也会破坏ANTLR将左递归规则转换为非左递归规则的机制。详见第14章）。

至此，我们已经看到了将词法符号放入不同通道的方法，以及将上下文相关的词法符号分解为更小的组件的方法。接下来，我们将会学习如何根据具体的上下文，将相同的字符序列处理为不同类型的词法符号。

3.有趣的Python换行符

对于开发者而言，Python的换行符处理机制是十分自然的。在Python中，语句的终止标志是换行符而非分号。我们中的大多数人只会在每行中放置一个语句，所以键入额外的分号只是徒劳而已。与此同时，我们不希望一行语句过长，所以Python在特定的上下文中会忽略换行符。例如，Python允许我们将一个函数调用分为多行，如：

```
f(1,  
2,  
3)
```

为了弄清楚究竟在什么时候需要忽略换行，让我们将Python的参考指南中所有有关换行的部分摘录出来，其中最重要的部分如下所示：

圆括号、方括号或者花括号中的表达式可以分散在多个物理行中。

所以，如果我们将表达式1+2中的+后插入一个换行符，Python就会报告一个错误。不过，(1+2)可以跨行。参考指南还指出，“隐式的续行可以带有注释”，以及“允许空白的续行”，如下所示：

```
f(1,  # 第一个参数  
  
2,    # 第二个参数  
      # 带有注释的空行  
3)    # 第三个参数
```

还可以使用反斜杠，将多行显式连接为一个逻辑行。

两个或多个物理行可以使用反斜杠字符（\）连接成一个逻辑行，方式如下：当一个物理行以一个不在字符串中或注释中的反斜杠结束时，它会和接下来的一行连接形成一个单独的逻辑行，反斜杠和后面的换行符会被删掉。

这意味着，即使不用括号，我们也可以将语句分散在多行中，如：


```
1+\n2
```

虽然手册没有明确说明，但是“物理行以.....的反斜杠结束”这句话暗示了，在\和换行符之间没有任何注释存在。

上述描述带来的结果是：语法分析器和词法分析器均需要有选择地保留和丢弃部分换行符。在之前对词法符号通道的学习中，我们知道，令语法分析器始终检查可选的空白字符不是一个好办法。这意味着，处理可选的换行符成为Python词法分析器的职责。这就变成了另一个语法上下文决定词法分析器行为的问题。

谨记上述规则，让我们一起编写一份识别简单的Python代码的语法，它能够匹配赋值语句和简单的表达式。我们将忽略字符串，以便专注于处理注释和换行符。下面是文法规则：

```
lexmagic/SimplePy.g4
```

```
file:  stat+ EOF ;

stat:  assign NEWLINE
      |  expr NEWLINE
      |  NEWLINE      // 忽略空行
      ;

assign: ID '=' expr ;

expr:  expr '+' expr
      |  '(' expr ')'
      |  call
      |  list
      |  ID
      |  INT
      ;

call:  ID '(' ( expr (',' expr)* )? ')' ;

list:  '[' expr (',' expr)* ']' ;
```

接下来处理词法分析器，让我们先编写几条熟悉的规则。匹配整数的INT规则已经很常见了，此外，根据参考指南，标识符的定义如下：

```
identifier ::= (letter|"_") (letter | digit | "_")*
letter      ::= lowercase | uppercase
```

使用ANTLR标记转写的结果如下：

```
lexmagic/SimplePy.g4
```

```
ID : [a-zA-Z_] [a-zA-Z_0-9]* ;
```

随后，我们需要一条常见的匹配空白字符的规则，以及一条匹配换行符的规则，后者会将NEWLINE词法符号输送给语法分析器。

```
lexmagic/SimplePy.g4
```

```
/** 终止语句的逻辑换行符 */  
NEWLINE  
    :   '\r'? '\n'  
    ;  
  
/** 注意：这里没有考虑 Python 的缩进规则 */  
WS   :   [ \t]+ -> skip  
    ;
```

为了处理Python的行注释，我们需要一条能够剥离注释但保留换行符的规则。

```
lexmagic/SimplePy.g4
```

```
/** 匹配注释。这里不匹配合换行符，因为我们需要将它送入语法分析器 */  
COMMENT  
    :   '#' ~[\r\n]* -> skip  
    ;
```

我们希望利用NEWLINE规则处理所有的换行符，使得：

```
i = 3 # assignment
```

能够被看作一个赋值语句后跟一个NEWLINE。

现在是时候处理特殊换行符的问题了。首先让我们从显式的行连接问题开始。我们新增了一条规则，匹配\后面紧跟着换行符的情况，并将这些字符丢弃。

```
lexmagic/SimplePy.g4
```

```
/** 忽略反斜杠换行符序列。这条规则不允许注释跟在反斜杠后面，  
 * 因为它后面必须是换行符  
 */  
LINE_ESCAPE  
:   '\\' '\r'? '\n' -> skip  
;
```

这意味着语法分析器不会看到上述字符序列。现在，我们要做的是，令词法分析器忽略括号中的换行符。这意味着我们需要一条名为 **IGNORE_NEWLINE** 的词法规则，它匹配类似 **NEWLINE** 的换行符，并且，如果该换行符处于括号中，则丢弃之。因为二者匹配的是相同的字符序列，存在歧义性，我们必须使用一个语义判定来区分它们。假设存在一个记录当前嵌套层数的变量 **nesting**，当词法分析器遇到了左括号但是没有遇到右括号时，该变量大于零，我们就可以写出如下的 **IGNORE_NEWLINE**:

```
lexmagic/SimplePy.g4
```

```
/** 嵌套在 (...) 或者 [...] 中的换行符将被忽略 */  
IGNORE_NEWLINE  
:   '\r'? '\n' {nesting>0}? -> skip  
;
```

此规则必须放置在 **NEWLINE** 之前，这样，当判定为真时，词法分析器就会按照解决歧义性的默认方法，选择 **IGNORE_NEWLINE** 规则。我们也可以将 **{nesting==0}? 判定放在 NEWLINE 中来达到同样的效果。**

在发现方括号和圆括号时，我们需要适当地调整此变量的值（我们的语法不支持花括号）。首先，我们需要定义该 **nesting** 变量。

```
lexmagic/SimplePy.g4
@lexer::members {
    int nesting = 0;
}
```

随后，我们需要在遇到括号时执行一些动作代码，从而增减`nesting`的值。如下列规则所示：

```
lexmagic/SimplePy.g4
LPAREN    : '(' {nesting++;} ;

RPAREN    : ')' {nesting--;} ;

LBRACK    : '[' {nesting++;} ;

RBRACK    : ']' {nesting--;} ;
```

严格而言，我们需要为方括号和圆括号各设置一个变量，以确保括号的精确匹配。不过，实际上我们无须担心类似`[1, 2)`这样的不匹配的括号，因为语法分析器会检测到该错误。在这样的语法错误中，对换行符的处理偏差是无关紧要的。

下列测试文件对Python的换行符和注释处理过程中的关键元素进行了测试：空行被忽略、在括号中的换行符被保留、反斜杠连同紧随其后的换行符被丢弃、括号中的注释不影响换行符的处理。

```
lexmagic/f.py
```

```
# 测试
f(1, # 第一个参数

    2, # 第二个参数
    # 空注释行
    3) # 第三个参数
g() # 尾部的注释

1+\
2+\
3
```

下面的构建和测试步骤将送给语法分析器的词法符号流中的NEWLINE进行了高亮显示:

```
$ antlr4 SimplePy.g4
$ javac SimplePy*.java
$ grun SimplePy file -tokens f.py
➤ [@0,8:8='\n',<11>,1:8]
  [@1,9:9='f',<4>,2:0]
  [@2,10:10='(',<6>,2:1]
  [@3,11:11='1',<5>,2:2]
  [@4,12:12=',',<1>,2:3]
  [@5,29:29='2',<5>,4:2]
  [@6,30:30=',',<1>,4:3]
  [@7,80:80='3',<5>,6:2]
  [@8,81:81=')',<7>,6:3]
➤ [@9,94:94='\n',<11>,6:16]
➤ [@10,95:95='\n',<11>,7:0]
  [@11,96:96='g',<4>,8:0]
  [@12,97:97='(',<6>,8:1]
  [@13,98:98=')',<7>,8:2]
➤ [@14,108:108='\n',<11>,8:12]
➤ [@15,109:109='\n',<11>,9:0]
  [@16,110:110='1',<5>,10:0]
  [@17,111:111='+',<2>,10:1]
  [@18,114:114='2',<5>,11:0]
  [@19,115:115='+',<2>,11:1]
  [@20,118:118='3',<5>,12:0]
```

➤ [`@21,119:119='\n',<11>,12:1]`
[`@22,120:119='<EOF>','<-1>,13:2]`

需要特别注意的是，在上面的词法符号流中有六个NEWLINE词法符号，但是f.py文件中存在十二个换行符。我们的词法分析器成功地剔除了六个换行符。其语法分析树如图12-5所示，其中换行符已经被高亮标示出来。

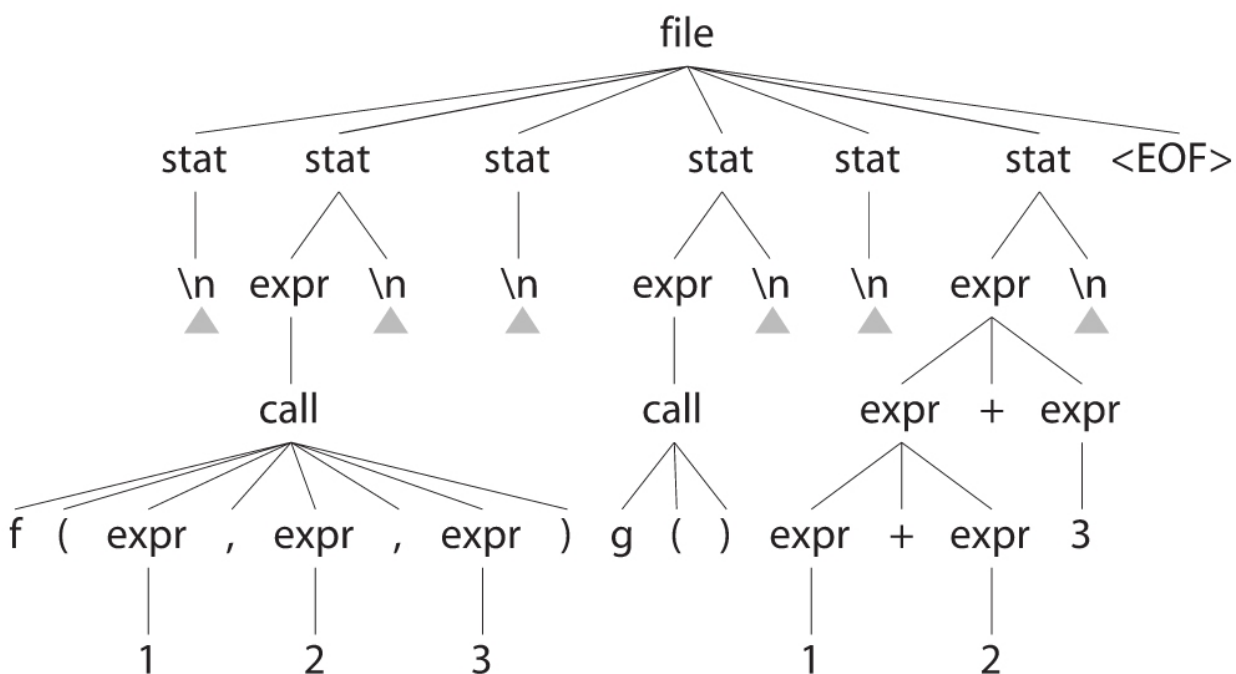


图12-5 将换行符高亮显示的语法分析树

第一个换行符是一个空行，被语法分析器识别为一个空语句（stat规则）。第三个和第五个换行符同样也是空语句。其余三个换行符都是表达式语句的终止符。使用Python解释器运行f.py（需要有适当的f（）和g（）定义）可知，f.py是合法的Python代码。

在上文中，我们解决了三种与词法符号有关的上下文相关问题。这里的上下文是由语义决定的，而非由输入文件的某个区域决定。接下来，我们计划研究这样的输入文件：其中一些独立的区域是我们所感兴趣的，它们被我们不关心的区域包围。

12.3 字符流中的孤岛

迄今为止，我们讨论的所有输入文件都只包含一种语言。例如，DOT、CSV、Python和Java的文件都只包含这些语言的文本。不过，还存在另外一些格式的文件，其中的结构化区域——或者称为孤岛——被随机的文本所包围。我们称这样的格式为孤岛语言（`island language`），并且使用孤岛语法（`island grammar`）来描述它们。孤岛语言的例子包括StringTemplate和LaTeX这样的模板引擎语言，不过以XML最为突出。在XML文件中，结构化的标签和&实体被大片我们不关心的文本所包围（由于各XML标签内部是结构化的，我们也可以称XML为群岛语言[`archipelago language`]）。

一些文本是否是孤岛语言通常取决于我们的观点。如果我们编写一个C预处理器，那么预处理命令就构成了孤岛语言，而C代码就是周围的“海洋”。而如果我们在为IDE编写C语言的语法分析器，那么它就必须忽略预处理命令构成的“海洋”。

在本节中，我们的目标是学习如何忽略“海洋”，并对“孤岛”进行词法分析，这样，语法分析器就能验证这些“孤岛”中的语法是否正确。在下一节中，我们需要这两种方式来编写一个真正的XML语法分析器。首先，我们来学习如何从XML文件的“海洋”中区分出“孤岛”。

为将XML的标签和普通文本区分开，我们首先想到的方案是编写一个处理输入字符流的过滤器，丢弃标签之间的全部内容。这样也许能够令词法分析器更加容易识别出孤岛部分，但过滤器会丢弃所有的普通文本内容——这是我们不希望看到的。例如，对于输入
<name>John</name>，我们并不希望丢弃John。

真正的解决方案是，首先编写一份子XML语法，它将标签内的文本识别为一种词法符号，标签外的文本识别为另一种词法符号。由于本章主要关注词法分析器，我们会使用一条规则来匹配一串标签、&实体、CDATA区域以及普通文本（即海洋部分）。

```
lexmagic/Tags.g4
grammar Tags;
file : (TAG|ENTITY|TEXT|CDATA)* ;
```

file规则并不验证XML的格式是否正确——它只识别XML中的各种词法符号。

为正确地分割XML文件，我们为孤岛部分指定了词法规则，而在最后放置了一条名为TEXT的规则来兜底，它匹配其余的任何内容。

```
lexmagic/Tags.g4
```

```
COMMENT : '<!--' .*? '-->' -> skip ;  
CDATA : '<![CDATA[' .*? ']]>';  
TAG : '<' .*? '>' ; // 必须放置在其他类似标签的结构之后  
ENTITY : '&' .*? ';' ;  
TEXT : ~[<&]+ ; // 除 < 和 & 之外的任意字符序列
```

上述规则大量使用了“.*?”非贪婪匹配（详见5.5节中“匹配字符串常量”部分），它会一直向后扫描，直至遇到匹配后续规则的内容为止。

TEXT规则匹配一个或多个字符，只要它们不是标签或者实体的起始字符即可。可能有人想要使用“.+”代替“~[<&]+”，那样的话，一旦进入了循环，它就会吞掉所有的输入字符。因为TEXT中的“.+”后面没有任何内容，所以该循环就无法停止。

这里使用了一个细微但重要的解决歧义性的方案。在2.3节中我们学到，ANTLR词法分析器解决歧义性的方法是选择语法文件中位置靠前的规则。例如，TAG规则匹配尖括号中的任意内容，而这也包括了注释和CDATA区域。由于我们的COMMENT和CDATA规则位置靠前，TAG规则就只能匹配到其他规则无法成功匹配的标签。

需要注意的是，技术上，XML不允许以“--->”结尾的注释和包含“--”的注释。使用我们在9.4节中学到的方法，我们可以为这些非法的注释编写词法规则，输出指定的错误消息。

```

BAD_COMMENT1:  '<!--' .*? '--->'
               {System.err.println("Can't have ---> end comment");} -> skip ;
BAD_COMMENT2:  '<!--' ('--'|.)?*? '--->'
               {System.err.println("Can't have -- in comment");}      -> skip ;

```

不过，为简单起见，我们暂时不把上述两条规则加入我们的Tags语法中。

现在，让我们看看我们的子XML语法是如何处理下列输入的：

```

lexmagic/XML-inputs/cat.xml
<?xml version="1.0" encoding="UTF-8"?>
<?do not care?>
<CATALOG>
<PLANT id="45">Orchid</PLANT>
</CATALOG>

```

下面是构建和测试步骤，使用了grun来打印词法符号：

```

$ antlr4 Tags.g4
$ javac Tags*.java
$ grun Tags file -tokens XML-inputs/cat.xml
[@0,0:37='<?xml version="1.0" encoding="UTF-8"?>',<3>,1:0]
[@1,38:38='\n',<5>,1:38]
[@2,39:53='<?do not care?>',<3>,2:0]
[@3,54:54='\n',<5>,2:15]
[@4,55:63='<CATALOG>',<3>,3:0]
[@5,64:64='\n',<5>,3:9]
[@6,65:79='<PLANT id="45">',<3>,4:0]
[@7,80:85='Orchid',<5>,4:15]
[@8,86:93='</PLANT>',<3>,4:21]
[@9,94:94='\n',<5>,4:29]
[@10,95:104='</CATALOG>',<3>,5:0]
[@11,105:105='\n',<5>,5:10]
[@12,106:105='<EOF>',<-1>,6:11]

```

该语法正确地读取了XML文件，并匹配了其中的孤岛部分和普通文本。但是，它并没有提取出标签中的内容供语法分析器检查。

使用词法模式处理上下文相关的词法符号

标签内外的文本实际上是不同的语言。例如，`id="45"`在标签外仅仅是普通文本，但是在标签内部则是三个词法符号。在某种意义上，我们希望XML词法分析器根据上下文使用不同的规则进行匹配。ANLTR提供了词法模式（**lexical mode**），允许词法分析器在不同的上下文中切换（模式）。在本节中，我们计划学习词法模式，利用它来改进上一节中的子XML语法，这样，它就能将标签中的各组成部分送给语法分析器。

词法模式允许我们将单个词法分析器分成多个子词法分析器。词法分析器会返回被当前模式下的规则匹配的词法符号。一门语言能够进行模式切换的一个重要要求是包含清晰的词法“哨兵”，它能够触发模式的来回切换，例如尖括号。换言之，模式的切换只依赖于词法分析器可以从输入文本中获得的信息，而不依赖于语义上下文。

为简单起见，我们先编写一份XML语法的子集，其中标签仅包含标识符而不包含属性。我们会使用默认模式匹配标签外的海洋部分，另外一个模式匹配标签内的内容。当词法分析器在默认模式下发现`<`时，它就应当切换到孤岛模式（即标签内部模式）并向语法分析器输出一个

标签起始词法符号。当词法分析器在内部模式下发现>时，它就应该切换回默认模式并输出一个标签结束词法符号。内部模式需要一些规则来匹配标识符和/。下列规则运用了上述策略：

```
lexmagic/ModeTagsLexer.g4
lexer grammar ModeTagsLexer;

// 默认模式下的规则（海洋部分）
OPEN  : '<'      -> mode(ISLAND) ;      // 切换到 ISLAND 模式
TEXT  : ~'<'+ ;      // 收集所有的文本

mode ISLAND;
CLOSE : '>'      -> mode(DEFAULT_MODE) ; // 回到 SEA 模式
SLASH : '/' ;
ID    : [a-zA-Z]+ ;      // 匹配标签中的 ID 并将其输送给语法分析器
```

OPEN和TEXT规则位于默认模式下。OPEN匹配单个<，使用词法分析器指令mode（ISLAND）来切换模式。之后，词法分析器就只会使用ISLAND模式下的规则进行工作。TEXT匹配任意非标签起始字符的序列。由于这些词法规则中不包含skip指令，因此所有的文本都会被匹配为词法符号送给语法分析器。

在ISLAND模式中，词法分析器匹配>、/和ID词法符号。当词法分析器发现>时，它会执行切换回默认模式的指令，该模式由Lexer类中的常量DEFAULT_MODE标识。这就是词法分析器来回切换模式的方法。

和Tags语法一样，这份稍大的XML语法子集对应的语法分析器能够匹配标签和文本块，改进之处在于，我们现在使用了tag规则来匹配独立的标签元素而非一个单独的标签词法符号。

```
lexmagic/ModeTagsParser.g4
```

```
parser grammar ModeTagsParser;
```

```
options { tokenVocab=ModeTagsLexer; } // 使用 ModeTagsLexer.g4 中的词法符号
```

```
file: (tag | TEXT)* ;
```

```
tag : '<' ID '>'
    | '<' '/' ID '>'
    ;
```

其中唯一令我们感到陌生的语法是**tokenVocab**选项。当我们的语法分析器和词法分析器位于不同文件中时，我们需要确保两个文件中的词法符号类型和词法符号名称一致。例如，词法分析器中的词法符号**OPEN**必须和语法分析器中的同名词法符号具有相同的词法符号类型。

接下来让我们构建这份语法，并使用一些**XML**样例输入来对它进行测试。

```
⇒ $ antlr4 ModeTagsLexer.g4 # 必须先进行这一步，以获得 ModeTagsLexer.tokens
```

```
⇒ $ antlr4 ModeTagsParser.g4
```

```
⇒ $ javac ModeTags*.java
```

```
⇒ $ grun ModeTags file -tokens
```

```

⇒⇒ Hello <name>John</name>
⇒⇒ E0F
⌞ [@0,0:5='Hello ',<2>,1:0]
  [@1,6:6='< ',<1>,1:6]
  [@2,7:10='name',<5>,1:7]
  [@3,11:11='>',<3>,1:11]
  [@4,12:15='John',<2>,1:12]
  [@5,16:16='< ',<1>,1:16]
  [@6,17:17='/',<4>,1:17]
  [@7,18:21='name',<5>,1:18]
  [@8,22:22='>',<3>,1:22]
  [@9,23:23='\n',<2>,1:23]
  [@10,24:23='<E0F>',<-1>,2:24]

```

词法分析器将<name>作为三个词法符号（索引值分别为1、2、3）送给语法分析器。还需要注意的是，海洋部分中的Hello能够匹配ISLAND模式下的ID规则，但是，因为词法分析器起始状态是默认模式，所以Hello被识别成了TEXT词法符号。你可以从索引值为0和2的词法符号在词法符号类型上的差异发现这一点：name是ID类型的词法符号（词法符号类型为5）。

我们希望在语法分析器而非词法分析器中匹配标签语法的另一个原因是，语法分析器在执行动作代码上具有的灵活性远远高于词法分析器。另外，语法分析器能够自动建立语法分析树，如图12-6所示。

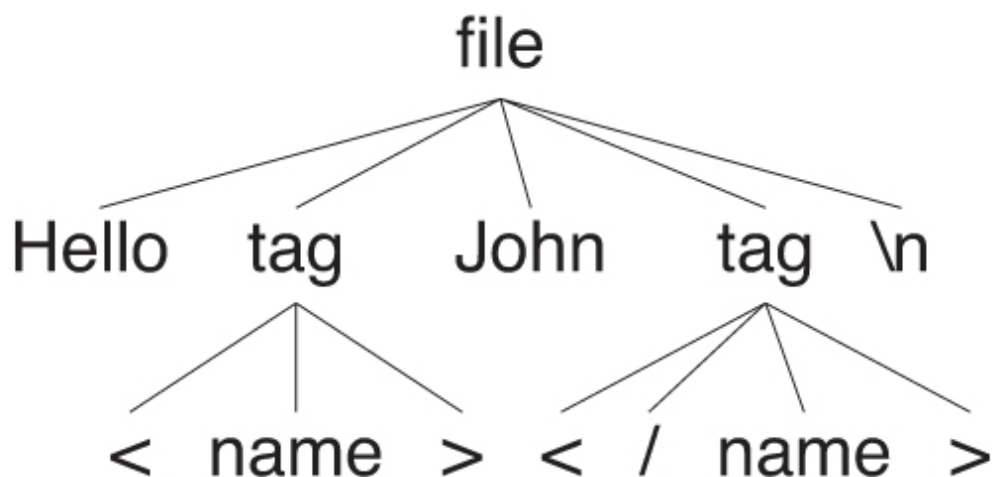


图12-6 语法分析器自动建立的语法分析树

为将语法应用于实际程序，我们既可以使用通常的监听器或者访问器机制，也可以为语法增加动作。例如，我们可以不建立树，而是使用语法的内嵌动作触发SAX方法调用来实现XML的SAX事件机制。现在，我们已经知道了如何将XML的海洋部分和孤岛部分分开，也了解了如何向语法分析器输送标签的各组成部分，下面我们将编写一个真正的XML语法分析器。

12.4 对XML进行语法分析和词法分析

由于XML是一门已经被严格定义的语言，在开始我们的XML工程之前，最好先研究一下W3C的XML语言定义。不幸的是，该XML规范（下面简称规范）巨大无比，很容易在浩如烟海的细节中迷失方向。为简单起见，我们会忽略掉在处理XML文件中不需要的东西：<!DOCTYPE..>文档类型定义（DTD），<! ENTITY>实体声明，以及

<! NOTATION..>符号声明。毕竟，处理上述标签教会我们的知识不会比处理其他结构多。

我们计划首先编写一些XML的语义规则。一个好消息是，我们可以将规范中的非正式语法规则逐字转写成ANTLR标记。

1.XML规范转换为ANTLR文法语法

运用之前的经验，我们能够很快地完成XML语法的编写。为避免遗漏，让我们仔细研究一下规范中的关键语法规则。

```
document      ::= prolog element Misc*
prolog        ::= XMLDecl? Misc*
content       ::= CharData?
               ((element | Reference | CDsect | PI | Comment) CharData?)*
element       ::= EmptyElemTag
               | STag content ETag
EmptyElemTag  ::= '<' Name (S Attribute)* S? '/>'
STag          ::= '<' Name (S Attribute)* S? '>'
ETag          ::= '</' Name S? '>'
XMLDecl       ::= '<?xml' VersionInfo EncodingDecl? SDDDecl? S? '?>'
Attribute     ::= Name Eq AttValue
Reference     ::= EntityRef | CharRef
Misc          ::= Comment | PI | S
```

我们还需要许多其他的规则，不过它们都是词法规则。这是一个很好的、展示如何在词法规则和文法规则间划分界线的例子。按照我们在5.6节中的讨论，需要遵循的关键准则是我们是否需要了解某个元素的内部细节。例如，我们不关心注释或者处理指令（Processing Instructions, PI），所以可以令词法分析器将其匹配为文本块。

下面让我们比较一下规范中的非正式规则与下列完整的ANTLR文法语法。与我们先前编写的其他语言——例如JSON和Cymbol——相比，XML的文法规则十分简单。

```
lexmagic/XMLParser.g4
parser grammar XMLParser;
options { tokenVocab=XMLLexer; }

document      :   prolog? misc* element misc*;

prolog        :   XMLDeclOpen attribute* SPECIAL_CLOSE ;

content       :   chardata?
                ((element | reference | CDATA | PI | COMMENT) chardata?)* ;

element       :   '<' Name attribute* '>' content '<' '/' Name '>'
                |   '<' Name attribute* '/>'
                ;

reference      :   EntityRef | CharRef ;

attribute     :   Name '=' STRING ; // 我们的 STRING 就是规范里的 AttValue
/** 其余所有未标记的文本构成了文档中的
 *  字符数据
 */
chardata      :   TEXT | SEA_WS ;

misc          :   COMMENT | PI | SEA_WS ;
```

规范中的规则和我们编写的规则存在诸多显著差异。首先，规范中的规则XMLDecl只能匹配三个特定属性（version、encoding和standalone），而我们的规则能够匹配<? xml...? >中的任意属性。稍后，将有一个语义分析的阶段来确保属性的名字是正确的。当然，我们也可以通过语法中的判定来解决此问题，但是那样会使语法的可读性和效率变差。

```
prolog      : XMLDecl versionInfo encodingDecl? standalone? SPECIAL_CLOSE ;
versionInfo : {_input.LT(1).getText().equals("version")}? Name '=' STRING ;
encodingDecl : {_input.LT(1).getText().equals("encoding")}? Name '=' STRING ;
standalone   : {_input.LT(1).getText().equals("standalone")}? Name '=' STRING ;
```

另外一个差别是，我们的词法分析器会匹配并丢弃标签中、属性间的空白字符，这样我们就无须在`element`规则中检查空白字符了（`element`是之前章节的`tag`规则的增强版）。我们的词法分析器还区分出了空白字符（`SEA_WS`）和标签外的非空白文本（`TEXT`），不过，它会将二者都当作词法符号输送给语法分析器（在之前章节中，标签外的所有文本都被当作一个单独的`TEXT`词法符号处理）。这是由于，规范虽然允许空白字符的存在，但在特定位置上不允许其出现，例如根元素前。因此，在我们的语法中，`chardata`是一条文法规则，而非一个词法符号。现在XML语法分析器已经基本可用，接下来我们将构建词法分析器，以获取更多的经验。

2.将XML词法符号化

通过从规范中提取所需的相关规则，我们就可以开始编写XML词法分析器了。

```

Comment    ::= '<!--' ((Char - '-') | ('-' (Char - '-')))* '-->'
CDsect     ::= '<![CDATA[' CData ']]>'
CData      ::= (Char* - (Char* ']]>' Char*)) // anything but ']]>'
PI         ::= '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
/** 除 'xml' 之外的任何名字 */
PITarget   ::= Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
/** 规范指出，CharData 是不包含任何标记的开始符和
 *   CDATA 区域结束符 "]]>" 的任意
 *   字符串
 */
CharData   ::= [^&]* - ([^&]* ']]>' [^&]*)
EntityRef  ::= '&' Name ';'
CharRef    ::= '&#' [0-9]+ ';'
           | '&#x' [0-9a-fA-F]+ ';'
Name       ::= NameStartChar (NameChar)*
NameChar   ::= NameStartChar | "-" | "." | [0-9] | #xB7
           | [#x0300-#x036F] | [#x203F-#x2040]

NameStartChar
    ::= ":" | [A-Z] | "_" | [a-z] | [#xC0-#xD6] | [#xD8-#xF6]
    | [#xF8-#x2FF] | [#x370-#x37D] | [#x37F-#x1FFF]
    | [#x200C-#x200D] | [#x2070-#x218F] | [#x2C00-#x2FEF]
    | [#x3001-#xD7FF] | [#xF900-#xFDCF] | [#xFDF0-#xFFFD]
    | [#x10000-#xEFFFF]
AttValue   ::= '"' ([^&"] | Reference)* '"'
           | "'" ([^&' ] | Reference)* "'"
S          ::= (#x20 | #x9 | #xD | #xA)+

```

真是复杂！我们会将它分解为三个不同的模式，并逐个编写之。我们需要的模式分别是标签外、标签内，以及特殊的<? ...? >标签内，这和我们在12.3节中“使用词法模式处理上下文相关的词法符号”部分中所做的非常相似。

比较规范中的规则和我们的词法规则，你可以发现，我们可以复用大多数规则的名字。虽然规范中描述规则的符号和ANTLR差别很大，但是我们可以从大多数规则中提取出核心思想。让我们从默认模式开

始，它匹配标签外的海洋部分。下面是我们的词法分析器的语法片段：

```
lexmagic/XMLLexer.g4
lexer grammar XMLLexer;

// 默认模式：标签外
COMMENT      :    '<!--' .*? '-->' ;
CDATA        :    '<![CDATA[' .*? ']]>' ;
/** 包括所有的 DTD、类似 <!ENTITY ...>
 * 的实体定义以及记号声明 <!NOTATION ...>
 */
DTD          :    '<!' .*? '>'          -> skip ;
EntityRef    :    '&' Name ';' ;
CharRef      :    '&#' DIGIT+ ';'
               |    '&#x' HEXDIGIT+ ';'
               ;
SEA_WS       :    '(' '|' '\t' '|' '\r'? '\n' ) ;

OPEN         :    '<'                      -> pushMode(INSIDE) ;
XMLDeclOpen  :    '<?xml' S                -> pushMode(INSIDE) ;
SPECIAL_OPEN:    '<?' Name                 -> more, pushMode(PROC_INSTR) ;

TEXT         :    '~[<&]+' ;                // 匹配任意除 < 和 & 之外的 16 位字符①
```

（注：即Java中的char。——译者注）

上述语法首先处理了可被视作完整的词法符号的词法结构，即COMMENT和CDATA词法符号。随后，我们匹配并丢弃了所有符合<! ...>形式的文档、实体和标记声明。在本例中，我们不关心它们。接下来是一些规则，用以匹配各种各样的实体和空白字符词法符号。最后是TEXT规则，它匹配除标签或实体的起始字符之外的任意输入文本。它实际上是一种特殊的“else语句”。

下面是最有趣的部分。当词法分析器发现标签的起始字符时，它就需要切换上下文，使得之后的词法符号被当作一个标签的合法部分处理，这就是OPEN规则完成的工作。和仅仅使用了mode指令的ModeTagsLexer语法不同，我们使用的是pushMode（以及稍后提到的popMode）。一旦将某种模式“进栈”，词法分析器就可以在未来将该模式“出栈”，从而返回到“调用者”对应的模式。它将在嵌套的模式切换中大显身手，不过我们暂时还没有用到它。

紧随其后的两个规则用于区分特殊的<? xml...? >标签和常规的<? ...? >处理指令。由于我们希望令语法分析器匹配<? xml...? >标签中的属性，词法分析器需要返回一个XMLDeclOpen词法符号并切换到INSIDE标签模式，在该模式下，属性词法符号会得到匹配。SPECIAL_OPEN规则匹配其他的<? ...? >标签并切换到PROC_INSTR模式（稍后我们将会看到）。它使用了一个不太常见的词法分析器指令more，它命令词法分析器寻找下一个词法符号，下一个词法符号将包含当前词法符号的文本。

当处于PROC_INSTR规则中时，我们通过IGNORE规则，令词法分析器将所有的字符放在一起，直到发现处理指令的结束字符“? >”为止。

```
lexmagic/XMLLexer.g4
```

```
mode PROC_INSTR;
PI      :    '>'          -> popMode ; // 关闭 <?...?>
IGNORE  :    .             -> more ;
```

这就是匹配除<? xml...? >标签之外的'<? !.*? '? >'处理指令的方法。SPECIAL_OPEN规则也能够匹配<? xml，但是由于位置靠前，XMLDeclOpen规则在词法分析器中的优先级更高，详见2.3节中的讨论。不过，只用一条'<? !.*? '? >'规则，然后在PROC_INSTR模式中处理所有的事情是行不通的。因为'<? !.*? '? >'匹配的字符序列要比'<? xml'S长得多，所以词法分析器永远不会匹配到XMLDeclOpen。这种情况和12.2节中“避免最长匹配带来的歧义性”部分中的词法分析器优先选择一个>>而非两个>相似。

注意，SPECIAL_OPEN引用了Name规则，而它并没有出现在现有的两种模式中。它位于我们稍后即将看到的INSIDE模式中。模式仅仅告诉词法分析器使用哪一组规则来匹配词法符号。因此，一条规则调用另一个模式中的规则是完全可行的。不过，需要记住的是，词法分析器仅能将当前词法模式所定义的词法符号类型返回给语法分析器。

最后一种模式是INSIDE模式，它识别标签内的所有元素，如：

```
title id="chap2", center="true"
```

标签内的词法结构再次证明了5.3节中的结论：从词法角度看，许多语言是相同的。例如，一个C语言的词法分析器能够毫不费力地对XML标签中的内容进行词法符号化。

下面就是处理标签内部结构的最后一种模式：


```
lexmagic/XMLLexer.g4
```

```
mode INSIDE;
```

```
CLOSE      :    '>'                                -> popMode ;
```

```
SPECIAL_CLOSE:  '?>'                                -> popMode ; // 关闭 <?xml...?>
```

```
SLASH_CLOSE :  '/>'                                -> popMode ;
```

```
SLASH       :  '/' ;
```

```
EQUALS      :  '=' ;
```

```
STRING      :  '"' ~[<"]* '"'  
              |  '\'' ~[<']* '\''  
              ;
```

```
Name       :  NameStartChar NameChar* ;
```

```
S          :  [ \t\r\n]                        -> skip ;
```

```
fragment
```

```
HEXDIGIT    :  [a-fA-F0-9] ;
```

```
fragment
```

```
DIGIT       :  [0-9] ;
```

```
fragment
```

```
NameChar    :  NameStartChar  
              |  '-' | '.' | DIGIT  
              |  '\u00B7'  
              |  '\u0300' .. '\u036F'  
              |  '\u203F' .. '\u2040'  
              ;
```

```
fragment
```

```
NameStartChar  
              :  [ :a-zA-Z]  
              |  '\u2070' .. '\u218F'  
              |  '\u2C00' .. '\u2FEF'  
              |  '\u3001' .. '\uD7FF'  
              |  '\uF900' .. '\uFDCF'  
              |  '\uFDF0' .. '\uFFFD'  
              ;
```

其中，开头三条规则匹配标签的结束序列，这就是`popMode`词法指令的用法。我们不需要指定目标模式，因为之前的模式位于栈顶，所以这些规则只需要“弹出”即可。`STRING`规则对应规范中的`AttValue`，唯一

的区别在于**STRING**不需要指定字符串中的实体。我们不关心字符串的内容，所以没有必要仔细匹配每个细节。我们只需按照规范的要求，确保字符串中不出现<和引号即可。

有了词法语法和文法语法后，我们就可以进行构建和测试了。

3.测试我们的XML语法

和之前一样，我们需要对这两个语法运行**ANTLR**，不过，需要先处理词法分析器，因为语法分析器依赖**XMLLexer.g4**生成的词法符号类型。

```
$ antlr4 XMLLexer.g4
$ antlr4 XMLParser.g4
$ javac XML*.java
```

下面是样例XML输入文件：

```
lexmagic/XML-inputs/entity.xml
<!-- a comment
-->
<root><!-- comment --><message>if salary < 1000</message>
&apos; <a>hi</a> <foo/>
</root>
```

使用**grun**生成语法分析树：

```
$ grun XML document -gui XML-inputs/entity.xml
```

如图12-7所示生成的语法分析树显示，我们的语法分析器正确地处理了注释、实体、标签和文本。

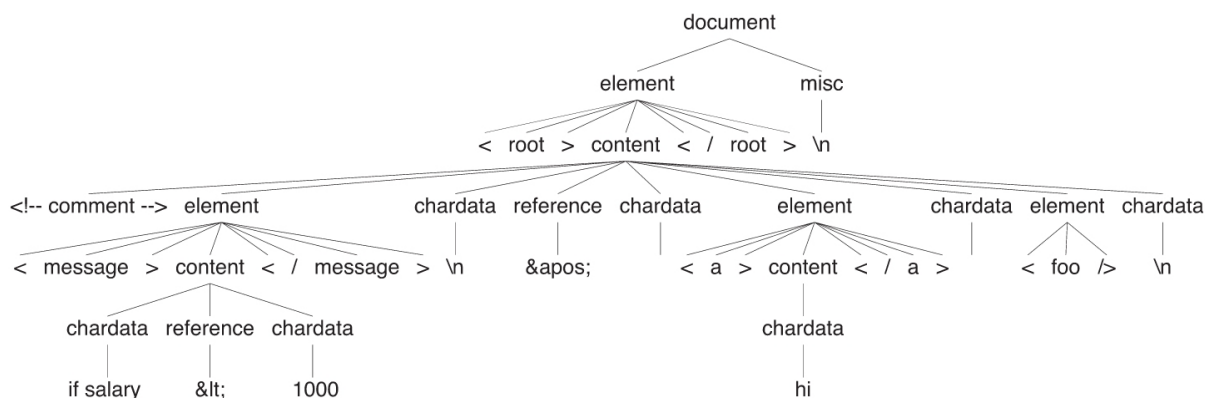


图12-7 正确处理注释、实体、标签和文本的语法分析树

接下来，我们需要确保我们的语法分析器能够正确处理`<? xml...? >`和其他的处理指令标签。下面是样例输入文件：

```
lexmagic/XML-inputs/cat.xml
<?xml version="1.0" encoding="UTF-8"?>
<?do not care?>
<CATALOG>
<PLANT id="45">Orchid</PLANT>
</CATALOG>
```

可以用以下指令生成语法分析树：

```
$ grun XML document -ps /tmp/t.ps XML-inputs/cat.xml
```

如图12-8所示语法分析树显示，输送给语法分析器的XML声明标签已被正确分片，而`<? do not care? >`以一个PI块的形式出现。

INSIDE模式下的大多数词法规则都使用了有效的Unicode码点，因而能够正确地匹配标签名。这允许我们识别国际化的XML文件——例如，

日语的标签。在我们的语法分析器中运行样例文件weekly-euc-jp.xml需要为grun正确地设置日语编码选项。

```
$ grun XML document -gui -encoding euc-jp XML-inputs/weekly-euc-jp.xml
```

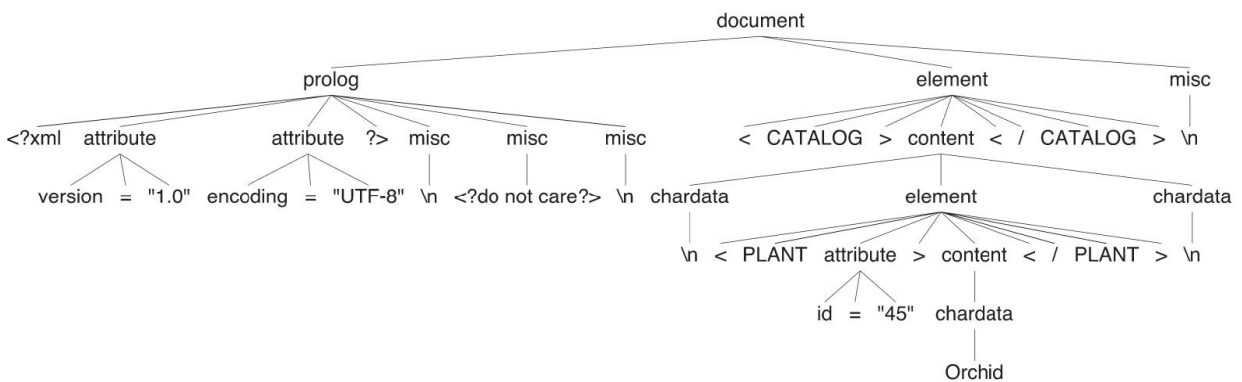


图12-8 XML声明标签被正确分片的语法分析树

如图12-9所示，它用一个大对话框显示了生成的结果。

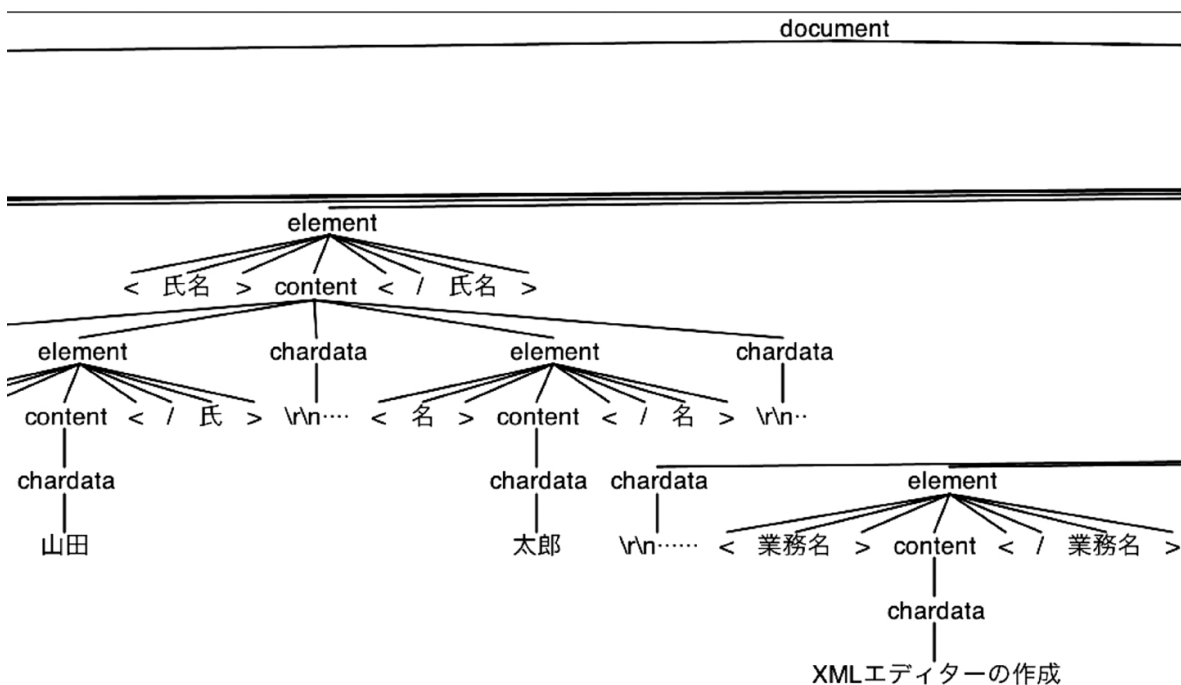


图12-9 一个对话框窗口

本章编写的XML语法用实例告诉我们，词法分析器常常具有相当的复杂度。相比之下，语法分析器通常大而简单。当一门语言难以识别时，通常是由于难以将字符组合成词法符号。这种情况的出现，一方面是因为词法分析器需要语义上下文来进行决策，另一方面是因为输入文件中存在多个遵循不同词法规则的区域。

本章篇幅较长，我们探讨了许多方法，希望你在遇到识别过程中的难题时将它们当作工具书使用。首先，我们学习了如何将不同的词法符号送入不同的通道，这样，我们就可以忽略但不丢弃类似注释和空白字符的关键词法符号。之后，我们研究了一些上下文相关的词法问题，例如棘手的“关键字作为标识符”问题。接下来，我们利用词法模式将输入文件的不同区域分别进行词法符号化，即将海洋部分和孤岛部分区分开。最后，我们使用词法模式完成了一个准确的XML词法分析器。

现在，我们对ANTLR用法的理解已经相当深刻。本书的下一个部分是参考章节，它填补了之前的行文中，出于连贯性目的而遗漏的一些细节。

**本书由“ePUBw.COM”整理，ePUBw.COM 提供
最新最全的优质电子书下载！！！！**

第四部分 ANTLR 参考文档

本书的前三部分是对ANTLR使用的指导，第四部分主要是参考文档。我们会首先总结运行时API，然后研究ANTLR对左递归规则的处理。最后，我们会看到庞大的索引章节。

第13章 探究运行时API

本章总结了ANTLR的运行时API，旨在帮助读者上手ANTLR运行库。它详细讲解了面向开发者的类，而这并非是复述Javadoc中的细节。请参阅类注释或方法注释，以了解其详细用法。

13.1 包结构概览

ANTLR运行时由六个包组成，其中，主要的包org.antlr.v4.runtime中的大多数类是面向应用程序的。在本书中，最常用到的是那些用于启动语法分析器分析输入文本的类。下列代码片段与一份名为X.g的语法协同工作，其中有一个名为MyListener的语法分析树监听器，它实现了XListener:

```
XLexer lexer = new XLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
XParser parser = new XParser(tokens);
ParseTree tree = parser.XstartRule();

ParseTreeWalker walker = new ParseTreeWalker();
MyListener listener = new MyListener(parser);
walker.walk(listener, tree);
```

我们第一次接触它是在3.3节中。

下面是对包的总结:

org.antlr.v4.runtime 该包包含了最常用的类和接口，例如与输入流、字符和词法符号缓冲区、错误处理、词法符号构建、词法分析和语法分析相关的类体系结构。

org.antlr.v4.runtime.atn 该包在ANTLR内部用于自适应LL（*）词法分析和语法分析策略。包名中的atn是增强转移网络（**argumented transition network**）的缩写，它是一种能够表示语法的状态机，其中网络的边代表语法元素。在词法分析和语法分析的过程中，ANTLR沿ATN移动，并基于前瞻符号作出预测。

org.antlr.v4.runtime.dfa 使用ATN进行决策的代价很高，因此ANTLR在运行过程中将预测结果缓存在了确定有限状态自动机（**Deterministic Finite Automata, DFA**）中。该包包含了所有的DFA实现类。

org.antlr.v4.runtime.misc 该包包含各种各样的数据结构，以及最常用的TestRig类——我们已经在之前章节中通过grun命令使用过它了。

org.antlr.v4.runtime.tree 默认情况下，ANTLR自动生成的语法分析器会建立语法分析树，该包包含实现此功能所需的全部类和接口。这些类和接口中还包括基本的语法分析树监听器、遍历器以及访问器机制。

org.antlr.v4.runtime.tree.gui ANTLR自带一个基本的语法分析树查看器，可通过inspect（）方法访问之。你也可以通过save（）方法将语法

分析树保存为PostScript格式。TestRig的“-gui”选项亦会启动该查看器。

剩下的章节将对按功能分组的运行时API进行描述。

13.2 识别器

ANTLR自动生成的词法分析器和语法分析器是Lexer和Parser的子类。Recognizer基类抽象了识别字符序列或词法符号序列中语言结构的概念。

识别器（Recognizer）的数据来源是IntStream，我们稍后会看到。

图13-1所示是相关的实现和继承关系（接口用斜体标示）。

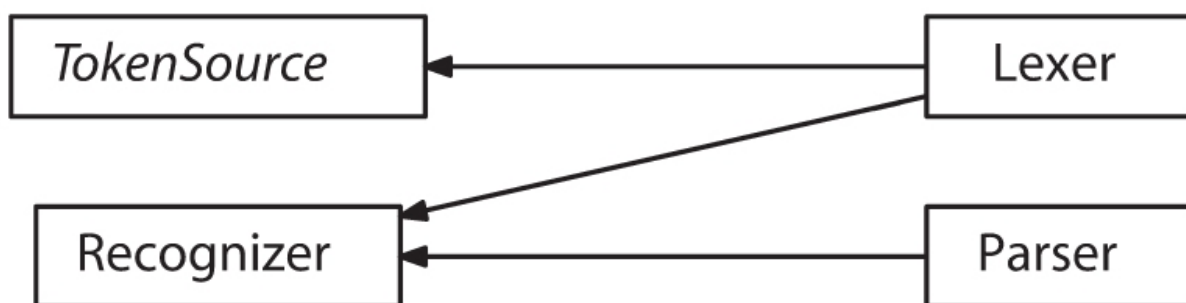


图13-1 相关的实现和继承关系

Lexer实现了接口TokenSource，后者包含两个核心的词法分析器方法：nextToken（）、getLine（）和getCharPositionInLine（）。按照一份ANTLR语法实现一个词法分析器并不十分困难。让我们编写一个词法分析器，用于将包含标识符和整数的下列输入文件进行词法符号化：


```
api/Simple-input
```

```
a 343x
```

```
abc 9 ;
```

手工编写的词法分析器的核心代码如下：

```

api/SimpleLexer.java
@Override
public Token nextToken() {
    while (true) {
        if ( c==(char)CharStream.EOF ) return createToken(Token.EOF);
        while ( Character.isWhitespace(c) ) consume(); // 丢弃空白字符
        startCharIndex = input.index();
        startLine = getLine();
        startCharPositionInLine = getCharPositionInLine();
        if ( c==';' ) {
            consume();
            return createToken(SEMI);
        }
        else if ( c>='0' && c<='9' ) {
            while ( c>='0' && c<='9' ) consume();
            return createToken(INT);
        }
        else if ( c>='a' && c<='z' ) { // 非常简单的 ID
            while ( c>='a' && c<='z' ) consume();
            return createToken(ID);
        }
        // error; consume and try again
        consume();
    }
}

protected Token createToken(int ttype) {
    String text = null; // 我们使用输入字符序列的 start..stop 子序列
    Pair<TokenSource, CharStream> source =
        new Pair<TokenSource, CharStream>(this, input);
    return factory.create(source, ttype, text, Token.DEFAULT_CHANNEL,
        startCharIndex, input.index()-1,
        startLine, startCharPositionInLine);
}

protected void consume() {
    if ( c=='\n' ) {
        line++; // \r 是一个普通字符, \n 会使得 line++
        charPositionInLine = 0;
    }
    if ( c!=(char)CharStream.EOF ) input.consume();
    c = (char)input.LA(1);
    charPositionInLine++;
}

```

如果使用手工编写的词法分析器，我们就需要一种方法，使之能和 ANTLR 语法共享词法符号名。为了让 ANTLR 自动地生成语法分析器代

码，我们需要令其知晓词法符号的类型整数值，这些值是在词法分析器的源代码中定义的。这就是.tokens文件的作用。

```
api/SimpleLexer.tokens
```

```
ID=1  
INT=2  
SEMI=3
```

下面是一份读取上述词法符号定义的简单语法：

```
api/SimpleParser.g4
```

```
parser grammar SimpleParser;  
options {  
    // 从 SimpleLexer.tokens 获取词法符号类型 don't name it  
    // 不要将它命名为 SimpleParser.tokens，因为 ANTLR 会覆盖它  
    tokenVocab=SimpleLexer;  
}  
  
s : ( ID | INT )* SEMI ;
```

下面是构建和测试的步骤：

```
$ antlr4 SimpleParser.g4  
$ javac Simple*.java TestSimple.java  
$ java TestSimple Simple-input  
(s a 343 x abc 9 ;)
```

(注：参见

<https://media.pragprog.com/titles/tpantlr2/code/api/TestSimple.java>。——译者注)

13.3 输入字符流和词法符号流

在最高层次的抽象中，词法分析器和语法分析器的主要工作都是分析整数输入流。词法分析器处理字符（短整数型），语法分析器处理词法符号类型（整数型）。这就是ANTLR有关输入流的类继承体系名为`IntStream`的原因。图13-2为这些类的继承体系。

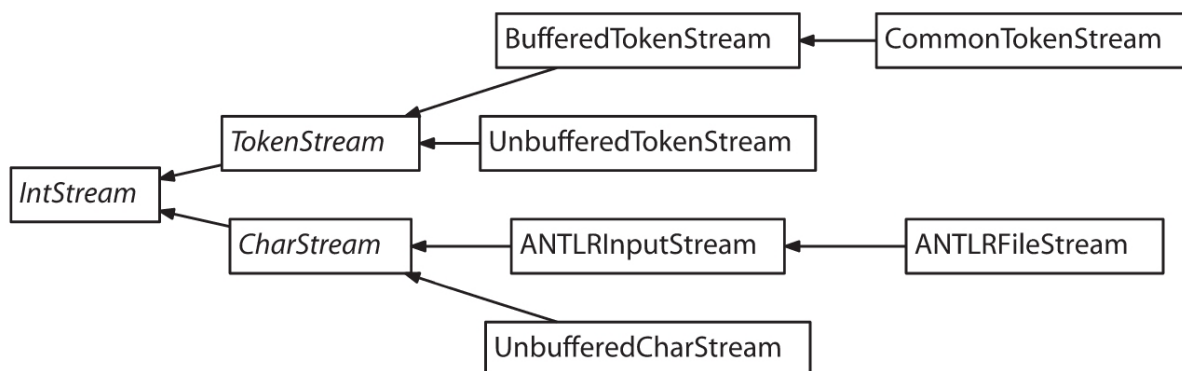


图13-2 ANTLR有关输入流的类继承体系

`IntStream`接口定义了流的大部分关键操作，包括消费符号以及获取前瞻符号的方法，即`consume ()`和`LA ()`。由于ANTLR中的识别器需要向前扫描并倒回原先的位置，`IntStream`还定义了`mark ()`和`seek ()`方法。

`CharStream`和`TokenStream`子接口增加了从流中提取文本的方法。实现它们的类通常会一次读取全部输入并将它们缓存起来。这种方案使得类的编写和访问输入更为容易，同时也更符合常见情况。如果输入过于庞大无法缓存，或者是无限长的（例如通过一个套接字），你可以使用`UnbufferedCharStream`和`UnbufferedTokenStream`。

进行语法分析的代码通常是：创建一个输入流，将一个词法分析器指定给该流，创建一个词法符号流并将其指定给该词法分析器，最后创建一个语法分析器并将其指定给该词法符号流。

```
ANTLRInputStream input = new ANTLRFileStream("an-input-file");  
//ANTLRInputStream input = new ANTLRInputStream(System.in); // 或者从标准输入读取  
SimpleLexer lexer = new SimpleLexer(input);  
CommonTokenStream tokens = new CommonTokenStream(lexer);  
SimpleParser parser = new SimpleParser(tokens);  
ParseTree t = parser.s();
```

13.4 词法符号和词法符号工厂

词法分析器将字符流分解成若干词法符号对象，语法分析器则尝试将语法结构应用于生成的词法符号流之上。通常，我们认为词法符号被词法分析器创建之后就不再改变，然而，有些时候，我们需要在创建词法符号之后修改它们的某些字段。例如，词法符号流在工作时会设置其中的词法符号的索引值。为提供对这些修改操作的支持，ANTLR使用WritableToken接口，它是一种带setter方法的词法符号。最后，我们得到了CommonToken，它是一个全功能的词法符号类，图13-3所示为Token的类继承体系。



图13-3 Token的类继承体系

通常情况下，我们无需实现自定义类型的词法符号。下面的示例代码展示了一个特殊的Token实现，它为每个词法符号添加了一个字段：

```
api/MyToken.java
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CommonToken;
import org.antlr.v4.runtime.TokenSource;
import org.antlr.v4.runtime.misc.Pair;

/** 一个用于追踪并保存 TokenSource 名字的 Token 实现类 */
public class MyToken extends CommonToken {
    public String srcName;

    public MyToken(int type, String text) {
        super(type, text);
    }

    public MyToken(Pair<TokenSource, CharStream> source, int type,
                  int channel, int start, int stop)
    {
        super(source, type, channel, start, stop);
    }

    @Override
    public String toString() {
        String t = super.toString();
        return srcName + ":" + t;
    }
}
```

为了让词法分析器生成这样的特殊词法符号，我们需要新建一个工厂对象，将其传给词法分析器。我们还需要通知语法分析器，使得它的错误处理器能够在必要时生成正确类型的词法符号，图13-4所示为工厂类TokenFactory的类继承体系。



图13-4 生成Token对象的TokenFactory工厂类及其实现

下面就是能够生成MyToken对象的工厂类：

```
api/MyTokenFactory.java
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.TokenFactory;
import org.antlr.v4.runtime.TokenSource;
import org.antlr.v4.runtime.misc.Interval;
import org.antlr.v4.runtime.misc.Pair;

/** 工厂类 TokenFactory 创建 MyToken 对象 */
public class MyTokenFactory implements TokenFactory<MyToken> {
    CharStream input;

    public MyTokenFactory(CharStream input) { this.input = input; }
    @Override
    public MyToken create(int type, String text) {
        return new MyToken(type, text);
    }
    @Override
    public MyToken create(Pair<TokenSource, CharStream> source, int type,
                          String text,
                          int channel, int start, int stop, int line,
                          int charPositionInLine)
    {
        MyToken t = new MyToken(source, type, channel, start, stop);
        t.setLine(line);
        t.setCharPositionInLine(charPositionInLine);
        t.srcName = input.getSourceName();
        return t;
    }
}
```

下列的示例代码展示了向词法分析器和语法分析器注册工厂类的过程：

```

api/TestSimpleMyToken.java
ANTLRInputStream input = new ANTLRFileStream(args[0]);
SimpleLexer lexer = new SimpleLexer(input);
➤ MyTokenFactory factory = new MyTokenFactory(input);
➤ lexer.setTokenFactory(factory);
CommonTokenStream tokens = new CommonTokenStream(lexer);

// 打印全部词法符号
tokens.fill();
List<Token> alltokens = tokens.getTokens();
for (Token t : alltokens) System.out.println(t.toString());

// 开始语法分析
SimpleParser parser = new SimpleParser(tokens);
➤ parser.setTokenFactory(factory);
ParseTree t = parser.s();
System.out.println(t.toStringTree(parser));

```

它复用了先前SimpleParser.g4的语法。下面是构建和测试的步骤:

```

$ antlr4 SimpleParser.g4
$ javac Simple*.java MyToken*.java TestSimpleMyToken.java
$ java TestSimpleMyToken Simple-input
Simple-input: [@0,0:0='a',<1>,1:0]
Simple-input: [@1,2:4='343',<2>,1:2]
Simple-input: [@2,5:5='x',<1>,1:5]
Simple-input: [@3,7:9='abc',<1>,2:1]
Simple-input: [@4,11:11='9',<2>,2:5]
Simple-input: [@5,13:13=';',<3>,2:7]
Simple-input: [@6,15:14='<EOF>',<-1>,3:1]
(s a 343 x abc 9 ;)

```

MyToken类中的toString () 方法在原先的词法符号输出之前增加了一个Simple-input: 前缀。

13.5 语法分析树

Tree接口定义了一棵包含数据和子节点的树。SyntaxTree是一种知道如何将TokenStream中的词法符号组装成树节点的树。更详细地,

ParseTree代表语法分析树中的一个节点。它能够返回自己的所有后代中叶子节点包含的文本。我们已经在2.4节中看到了语法分析树的例子，以及不同类型的树节点对应的类。**ParseTree**也在**ParseTreeVisitor**中提供了常用的访问器模式的双分派方法**accept ()**，详见2.5节。图13-5所示为**Tree**接口的类继承体系。

RuleNode和**TerminalNode**对应着子树的根节点和叶子节点。**ANTLR**在单词法符号补全的恢复过程中会创建**ErrorNodeImpl**节点（详见9.3节中的“从不匹配的词法符号中恢复”部分）。

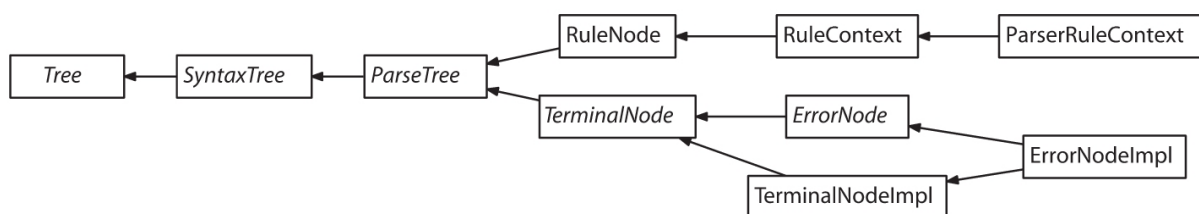


图13-5 **Tree**接口的类继承体系

RuleContext对象记录了一条规则的调用过程，通过**getParent ()**链，我们可以获得调用的上下文。**ParserRuleContext**包含一个字段，用于在语法分析器建立新子树时追踪其子节点。它们是树节点的主要实现类，**ANTLR**基于它们，为你的语法中的每条规则生成一个特殊的子类，你可以仔细查看它们。

13.6 错误监听器和监听策略

与ANTLR的语法错误处理机制相关的关键接口有两个：

`ANTLRErrorListener`和`ANTLRErrorStrategy`。我们已经在9.2节中学习了前者，在9.5节中学习了后者。监听器允许我们修改错误消息和输出的位置。我们可以通过实现不同的策略，来改变语法分析器应对错误的方式。图13-6所示为二者的类继承体系。

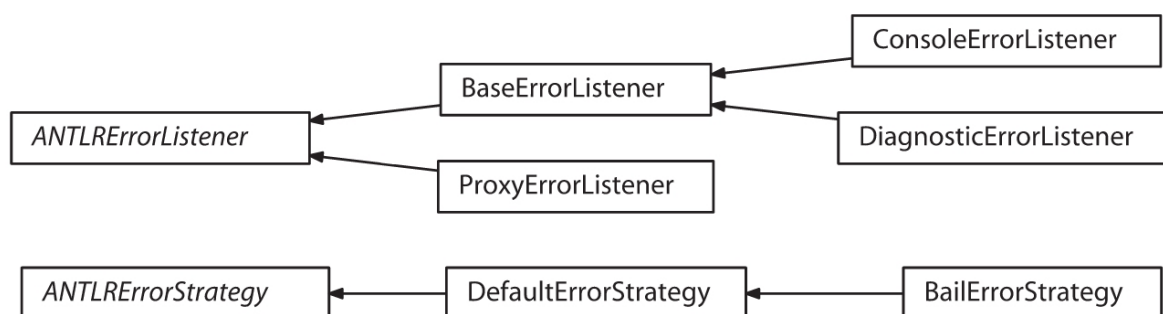


图13-6 `ANTLRErrorListener`和`ANTLRErrorStrategy`的类继承体系

ANTLR根据错误的具体类型，抛出特定的`RecognitionException`。需要注意的是，它们是不受检的运行时异常（unchecked runtime exception），所以你无须在方法上书写大量的`throws`语句，图13-7所示为`RecognitionException`的类继承体系。

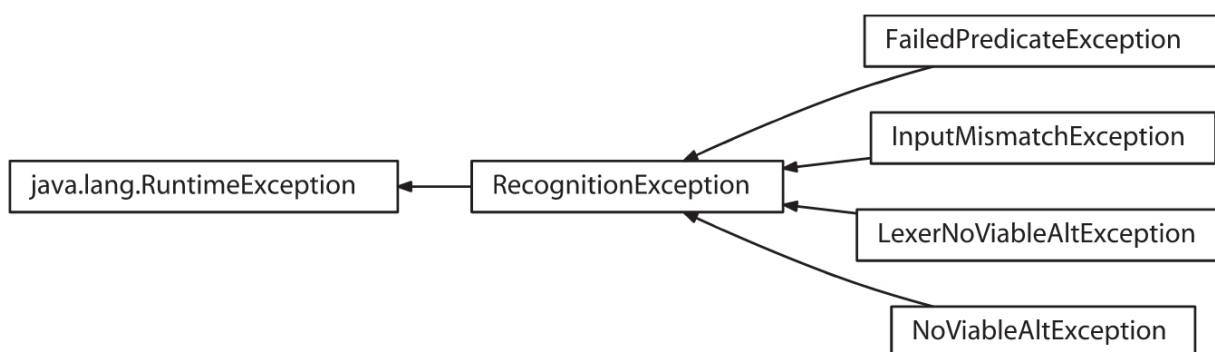


图13-7 RecognitionException的类继承体系

13.7 提高语法分析器的速度

ANTLR 4的自适应语法分析策略功能比ANTLR 3更加强大，不过这是以少量的性能损失为代价的。如果你需要尽可能快的速度和尽可能少的内存占用，你可以使用两步语法分析策略。第一步使用功能稍弱的语法分析策略——SLL (*)——在大多数情况下它已经足够了（它和ANTLR 3的策略相似，只是不需要回溯）。如果第一步的语法分析失败，那么就必须使用全功能的LL (*) 语法分析。这是因为，在第一步失败后，我们无法知道原因究竟是真正的语法错误，还是SLL (*) 的功能不够强大。由于能够通过SLL (*) 的输入一定能够通过全功能的LL (*) ，所以一旦第一步成功，就无须使用更昂贵的策略。

```

parser.getInterpreter().setSLL(true); // 尝试简单快捷的 SLL(*)
// 在第一次尝试过程中, 无需错误消息和错误恢复
parser.removeErrorListeners();
parser.setErrorHandler(new BailErrorStrategy());
try {
    parser.startRule();
    // 如果抵达此处, 证明没有语法错误, SLL(*) 就够了
    // 无需使用全功能的 LL(*)
}
catch (RuntimeException ex) {
    if (ex.getClass() == RuntimeException.class &&
        ex.getCause() instanceof RecognitionException)
    {
        // BailErrorStrategy 会将 RecognitionExceptions 封装在
        // RuntimeException 中, 所以这里需要检查是不是
        // 一个真正的 RecognitionException
        tokens.reset(); // 回滚输入流
        // 重新使用标准的错误监听器和错误处理器
        parser.addErrorListener(ConsoleErrorListener.INSTANCE);
        parser.setErrorHandler(new DefaultErrorStrategy());
        parser.getInterpreter().setSLL(false); // 尝试全功能的 LL(*)
        parser.startRule();
    }
}

```

如果第二步失败, 那就意味着一个真正的语法错误。

13.8 无缓冲的字符流和词法符号流

因为ANTLR的识别器在默认情况下会将输入的完整字符流和全部词法符号放入缓冲区, 所以它无法处理大小超过内存的文件, 也无法处理类似套接字 (socket) 连接之类的无限输入流。为解决此问题, 你可以使用字符流和词法符号流的无缓冲版本: `UnbufferedCharStream`和 `UnbufferedTokenStream`, 它们使用一个滑动窗口来处理流。

为展示二者的实际应用, 下列语法是6.1节中CSV语法的变体, 它计算一个文件中两列浮点数的和:

api/CSV.g4

```
/** 每行是两个实数:
    0.9962269825793676, 0.9224608616182103
    0.91673278673353, -0.6374985722530822
    0.9841464019977713, 0.03539546030010776
    ...
*/
grammar CSV;

@members {
double x, y; // 在这两个字段中保存列的和
}

file: row+ {System.out.printf("%f, %f\n", x, y);} ;

row : a=field ',' b=field '\r'? '\n'
    {
        x += Double.valueOf($a.start.getText());
        y += Double.valueOf($b.start.getText());
    }
    ;

field
    : TEXT
    ;

TEXT : ~[,\\n\\r]+ ;
```

如果你需要的只是每一列的和，你就应该在内存中只保留一个或两个词法符号用于记录结果。欲关闭ANTLR的缓冲功能，需要完成三件事情。首先，使用无缓冲的流代替常见的ANTLFileStream和CommonTokenStream。其次，传给词法分析器一个词法符号工厂，将输入流中的字符拷贝到生成的词法符号中去。否则，词法符号的getText（）方法就会尝试访问可能已经不再可用的字符流（详见2.4节中的图，它显示了词法符号和字符流的关系）。最后，阻止语法分析器建立语法分析树。下面的测试代码将关键行予以高亮显示：

api/TestCSV.java

```
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CommonToken;
import org.antlr.v4.runtime.CommonTokenFactory;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.UnbufferedCharStream;
import org.antlr.v4.runtime.UnbufferedTokenStream;

import java.io.FileInputStream;
import java.io.InputStream;
public class TestCSV {
    public static void main(String[] args) throws Exception {

        String inputFile = null;
        if ( args.length>0 ) inputFile = args[0];
        InputStream is = System.in;
        if ( inputFile!=null ) {
            is = new FileInputStream(inputFile);
        }
        > CharStream input = new UnbufferedCharStream(is);
        > CSVLexer lex = new CSVLexer(input);
        // 将滑动缓冲区中的文本拷贝到词法符号中
        > lex.setTokenFactory(new CommonTokenFactory(true));
        > TokenStream tokens = new UnbufferedTokenStream<CommonToken>(lex);
        CSVParser parser = new CSVParser(tokens);
        > parser.setBuildParseTree(false);
        parser.file();
    }
}
```

下面是使用一个1000行的样例文件进行构建和测试的步骤:

```
$ antlr4 CSV.g4
$ javac TestCSV.java CSV*.java
$ wc sample.csv
  1000    2000   39933 sample.csv # 1000 行 , 2000 个单词 , 39933 个字符
$ java TestCSV sample.csv
1000.542053, 1005.587845
```

为验证它是无缓冲的, 我使用了一个包含780万行记录、310M的CSV文件进行测试, 同时将JVM的RAM限制为10M大小。

```
$ wc big.csv
7800000 15600000 310959090 big.csv # 7800000 lines, ...
$ time java -Xmx10M TestCSV big.csv
11695395.953785, 7747174.349207

real    0m43.415s # 计算耗时
user    0m51.186s
sys     0m6.195s
```

当效率是首要目标时，无缓冲流是非常有用的（你可以将它们与先前章节中的各种技术联合使用）。使用它们的缺点是你需要手工处理与缓冲区相关的事情。例如，你不能在规则的内嵌动作中使用`$text`，因为它们是从输入流中获取文本的。

13.9 修改ANTLR的代码生成机制

ANTLR使用两种辅助工具来生成代码：一组StringTemplate文件（包含模板）以及一个称为LanguageTarget的Target子类，其中Language是语法的language选项。对应的StringTemplate组文件是 `org/antlr/v4/tool/templates/codegen/Language.stg`。例如，若希望修改Java的代码生成模板，你需要做的只是拷贝并修改 `org/antlr/v4/tool/templates/codegen/Java.stg`，然后，将它放在ANTLR的jar包之前的CLASSPATH中。ANTLR使用资源加载器（resource loader）来获取这些模板，这样，它就能首先找到你修改后的版本。

模板仅仅用于生成特定语法对应的代码，而大多数的常用功能位于运行库中。所以，`Lexer`和`Parser`等都是运行库的一部分，而非由ANTLR

生成。

欲增加对L语言的支持，你需要首先编写LTarget类，然后将其放置于org.antlr.v4.codegen包中，并让它在CLASSPATH中位于ANTLR的jar包之前。不过，只有在需要修改Target中的基础功能时，你才需要如此做。如果没有找到LTarget类，ANTLR就使用Target作为基础类（这也是ANTLR处理Java语言的方式）。

第14章 移除直接左递归

在5.4节中我们看到，用自然方式处理算术表达式是具有歧义性的。例如，下列expr可以将 $1+2*3$ 解释为 $(1+2)*3$ 或者 $1+(2*3)$ 。通过优先选择位置靠前的备选分支，ANTLR优雅地解决了歧义问题。

```
left-recursion-removal/Expr.g4
```

```
stat: expr ';' ;
```

```
expr:  expr '*' expr    // 优先级 4
      |  expr '+' expr   // 优先级 3
      |  INT             // 主表达式 ( 优先级 2)
      |  ID              // 主表达式 ( 优先级 1)
      ;
```

expr规则仍然是左递归的，传统的自顶向下的语法（例如ANTLR 3）无法处理这样的规则。在本章中，我们会探究ANTLR处理左递归和运算符优先级的方式。简单而言，ANTLR将左递归替换成一个 $(...)*$ ，它会比较前一个和下一个运算符的优先级。

熟悉这样的规则变换是很重要的，因为生成的代码反映的是转换后的规则，而非原先的规则。更重要的是，当一份语法没有按照我们的期望对运算符进行分组和结合时，我们需要知道原因。大多数用户可以只阅读本章第一节中与有效递归备选分支模式相关的内容，对实现细节感兴趣的进阶用户可以继续阅读第二节。

让我们首先学习ANTLR采取的转换方案，然后通过一个例子来在实践中学习优先级上升（precedence climbing）算法。

14.1 直接左递归备选分支模式

ANTLR通过检查下列四种子表达式运算模式来认定一条规则为左递归规则。

二元 `expr` 规则的某个备选分支符合 `expr op expr` 或者 `expr`

`(op1|op2|...|opN) expr` 的形式。`op` 可以是单一词法符号或者多词法符号构成的运算符。例如，Java语法可能独立处理尖括号，而非将`<=>`或`>=`当作单一词法符号。下面的备选分支将比较运算符按照同一优先级处理：

```
expr: ...  
    | expr ('<' '=' | '>' '=' | '>' | '<') expr  
    ...  
    ;
```

op可以是对另外一条规则的引用，例如，我们可能将若干个词法符号提出来，组成一条新的规则。

```
expr: ...  
    | expr compareOps expr  
    ...  
    ;  
compareOps : ('<' '=' | '>' '=' | '>' | '<') ;
```

三元 **expr**的某个备选分支符合**expr op1 expr op2 expr**的形式。**op1**和**op2**必须是单词法符号引用。这种模式的典型代表是类C语言中的“?:”运算符:

```
expr: ...  
    | expr '?' expr ':' expr  
    ...  
    ;
```

一元前缀 **expr**的某条规则符合**elements expr**的形式。ANTLR将任意元素后的尾递归规则引用视作一元前缀模式，前提是它不符合二元模式和三元模式。下面是两个具有前缀运算符的备选分支:

```
expr: ...  
    | '(' type ')' expr  
    ...  
    | ('+' | '-' | '++' | '--') expr  
    ...  
    ;
```

一元后缀 `expr`的某个备选分支符合`expr elements`形式。和前缀模式相同，**ANTLR**将任意元素前的直接左递归规则视作一元后缀模式，前提是它不符合二元模式和三元模式。下面是两个具有后缀运算符的备选分支：

```
expr: ...
    | expr '.' Identifier
    ...
    | expr '.' 'super' '(' exprList? ')'
    ...
    ;
```

其他形式的备选分支都被作为主表达式（**primary expression**）元素处理，例如标识符或者整数，也包括类似'(' `expr` ')'的形式，因为它不符合上述四种模式的任何一种。这是必要的，因为括号存在的意义是将其包含的表达式当作一个原子元素处理。这样的“其他形式”备选分支可以以任意顺序出现。**ANTLR**能够正确地处理它们。除此之外的备选分支顺序都是需要特别注意的。下面是一些主表达式备选分支的示例：

```
expr: ...
    | literal
    | Identifier
    | type '.' 'class'
    ...
    ;
```

除非额外指定，**ANTLR**假设所有的运算符都是左结合的。换句话说，`1+2+3`会被分组为 `(1+2)+3`。不过，某些运算符是右结合的，例如赋

值运算符和指数运算符，我们已经在5.4节中见过它们的处理方式。通过`assoc`选项，可以指定右结合性。

```
expr: expr '^'<assoc=right> expr
    ...
    | expr '='<assoc=right> expr
    ...
    ;
```

在下一节中，我们将会看到ANTLR翻译这些模式的方法。

14.2 左递归规则转换

如果你打开ANTLR命令行的“-Xlog”选项，你就可以在日志文件中看到转换后的左递归规则。下面是在先前Expr.g4语法中的`stat`和`expr`规则上发生的转换过程：

```
// 使用 "antlr4 -Xlog Expr.g4" 查看转换后的规则
stat:  expr[0] ';' ; // 匹配包含优先级运算符的表达式

expr[int _p]          // _p 是预期的最低优先级
:  ( INT              // 匹配主表达式（无运算符的表达式）
    | ID
    )
    // 匹配优先级大于等于预期最低值的运算符
    ( {4 >= $_p}? '*' expr[5] // * 具有优先级 4
      | {3 >= $_p}? '+' expr[4] // + 具有优先级 3
    )*
;
```

这些转换真是一项浩大的工程。不要被ANTLR处理`expr`的过程吓到，我们希望学习的是这些判定根据运算符优先级指导语法分析器进行正

确分组的方法。

关键在于，究竟是在`expr`的当前调用中匹配下一个运算符，还是令`expr`的调用者匹配下一个运算符。（...）*能够匹配当前运算符和右侧运算符。例如，对于输入`1+2*3`，该循环能够匹配`+2`和`*3`。循环中的判定能够决定，令语法分析器匹配这二者，还是放弃它们。如果操作符的优先级`3`低于当前子表达式预期的最低优先级`_p`，`{3>=$_p}`？就会关闭这个备选分支。

这不是带运算符优先级的语法分析

不要将这种机制和你可能在维基百科读到的带运算符优先级的语法分析相混淆。带运算符优先级的语法分析无法处理一些特殊情况，例如具有两种不同优先级的负号，一种用于取负，一种用于二元减法。它也无法处理具有两条相邻的规则备选分支，如`expr ops expr`。参考文献【**Compilers: Principles, Techniques, and Tools[ALSU06]**】给出了详细解释。

参数`_p`的值总是前一个运算符的优先级。`_p`从0开始，因为对`expr`的非递归调用会传递0，例如`stat`会调用`expr[0]`。为了解实际情况中`_p`的值，我们可以查看基于转换后的规则生成的语法分析树（参数`_p`的值在方括号中显示）。注意，这些语法分析树并不是ANTLR基于原先的左递

归规则建立的。这些是转换后的规则对应的语法分析树。如图14-1所示为样例输入和相应的语法分析树。

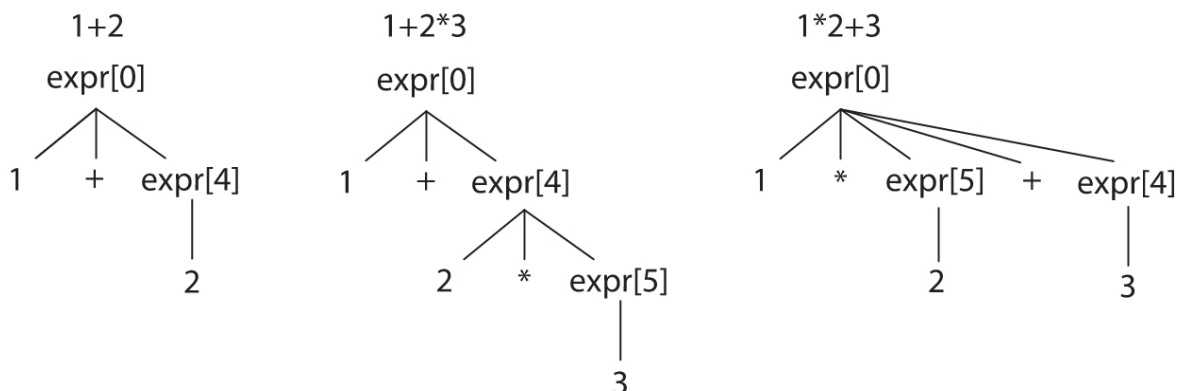


图14-1 样例输入及其相应的语法树

在第一棵树中，对`expr`的初始调用传递的`_p`是0，`expr`立刻匹配了

(`INT|ID`)子规则对应的1。现在`expr`必须决定是匹配接下来的`+`，还是直接退出循环并返回。此时，执行的判定是`{3>=0}`?，因此，我们进入了循环，匹配到了`+`，然后递归调用了`expr`规则，传递了参数4。下一次调用匹配到了2并立即返回，因为没有更多的输入了。`expr[0]`随后返回到了最初的`stat`中对`expr`的调用。

第二棵树展示了`expr[0]`匹配1，以及又一次的`{3>=0}`?判定，它允许我们匹配`+`和下一次调用`expr[4]`。这次调用匹配到了2，然后执行判定`{4>=4}`?，它允许语法分析器通过`expr[5]`，进一步匹配之后的`*`。

第三棵语法分析树是最有趣的。最初的调用`expr[0]`匹配了`1`和`*`，因为`{4>=0}`? 结果为真。此循环递归调用了`expr[5]`，并匹配了`2`。现在，在`expr[5]`的内部，语法分析器不应当匹配`+`，因为这样的话，`2+3`就会在乘法之前被执行（即在语法分析树中，我们会看到`expr[5]`的子节点是`2+3`）。判定`{3>=5}`? 关闭了对应的备选分支，因此`expr[5]`没有匹配`+`就提前返回了。在返回之后，由于`{3>=0}`? 为真，`expr[0]`匹配了`+3`。

我希望本章的内容能够加深大家对优先级上升机制的理解。欲了解更多细节，请参阅Norvell的论述。

第15章 语法参考

本书的大部分内容是ANTLR的使用指南。本章是对ANTLR语法及其关键语义的参考和总结。但是这并不意味着它是对ANTLR的使用方法的一份完整描述。本书中的所有示例的源代码都可以在网站上获取。

15.1 语法词汇表

ANTLR的词汇表为绝大多数开发者所熟知，因为它遵循C语言及其继承者的句法规则，此外，它还引入了一些扩展，用于对语法进行描述。

1. 注释

ANTLR支持单行、多行，以及Javadoc风格的注释。

```
/** 这份语法用于展示三种
 * 注释
 */
grammar T;

/* 多行
   注释
 */

/** 此规则匹配自定义语言中的一个声明 */
decl : ID ; // 匹配一个变量名
```

其中，Javadoc风格的注释不会被忽略，它们会被送入语法分析器。它们只能出现在语法和任意规则的开头。

2.标识符

词法符号名和词法规则名总是以大写字母开头，其中的“大写字母”由Java的Character.isUpperCase () 方法定义。文法规则总是以小写字母开头（即Character.isUpperCase () 方法返回值为false）。首字母之后的字符可以是大小写字符、数字和下划线。下面是一些样例：

```
ID, LPAREN, RIGHT_CURLY // 词法符号和词法规则名
expr, simpleDeclarator, d2, header_file // 文法规则名
```

与Java类似，ANTLR允许标识符中出现Unicode字符。

```
grammar 外;
a: '外';
```


ANTLR使用下列规则，以支持文法规则和词法规则中的Unicode字符：

```
ID : a=NameStartChar NameChar*
    {
        if ( Character.isUpperCase(getText().charAt(0)) ) setType(TOKEN_REF);
        else setType(RULE_REF);
    }
;
```

其中的NameChar即有效的标识符字符：

```
fragment
NameChar
:   NameStartChar
|   '0'..'9'
|   '_'
|   '\u00B7'
|   '\u0300'..' '\u036F'
|   '\u203F'..' '\u2040'
;
```

NameStartChar是能够作为标识符（规则、词法符号、标签名）首字符的字符列表。

```
fragment
NameStartChar
:   'A'..'Z' | 'a'..'z'
|   '\u00C0'..' '\u00D6'
|   '\u00D8'..' '\u00F6'
|   '\u00F8'..' '\u02FF'
|   '\u0370'..' '\u037D'
|   '\u037F'..' '\u1FFF'
|   '\u200C'..' '\u200D'
|   '\u2070'..' '\u218F'
|   '\u2C00'..' '\u2FEF'
```

```
|   '\u3001' .. '\uD7FF'  
|   '\uF900' .. '\uFDCF'  
|   '\uFDF0' .. '\uFFFD'  
;
```

这些字符大致与Java的Character类中的isJavaIdentifierPart () 和 isJavaIdentifierStart () 方法一致。如果你的语法文件编码不是UTF-8, 请确保在ANTLR工具中使用-encoding选项, 以便ANTLR能够正确地读取字符。

3. 文本常量

与大多数其他语言一样, ANTLR不区分字符常量和字符串常量。所有的文本常量都是由单引号括起来的字符串, 如'; '、'if'、'>='和\" (只包含一个单引号的字符串)。文本常量不支持正则表达式。

文本常量可以包含\uXXXX形式的Unicode转义序列, 其中XXXX是十六进制的Unicode字符值。例如, '\u00E8'是法语字符'è'。ANTLR也能够识别常见的转义序列: '\n' (换行符)、'\r' (回车符)、'\t' (制表符)、'\b' (退格符)、'\f' (换页符)。你可以直接使用它们或者使用Unicode的转义形式。详见code/reference/Foreign.g4。

ANTLR生成的识别器假定语法中的字符都是Unicode字符。ANTLR运行库根据目标语言对输入文件的编码作出假设。例如, 对于目标语言

是Java的情况，运行库假定输入文件为UTF-8编码。你可以使用一些类的构造器来指定编码，例如ANTLRFileStream。

4.动作

动作是使用目标语言编写的代码块。你可以在语法中的很多位置使用动作，它们的格式是相同的：由花括号包围的任意文本。如果右花括号位于字符串或者注释中，则无须转义它，如{"}"}或者{/*}*/; }。在花括号平衡的情况下，无须转义}，如{{...}}。其他情况下，额外的花括号需要使用反斜杠转义：{\}或者\}}。动作代码应当遵循language选项指定的目标语言的语法。

内嵌代码可以出现在以@header和@members命名的动作、词法和文法规则、指定异常捕获区、文法规则的属性区域（返回值、参数以及局部变量），以及一些规则元素的选项（当前只有判定）中。

ANTLR对动作进行解释的唯一情形是在语法的属性中，请参阅15.4节中“词法符号属性”部分和第10章。内嵌在词法规则中的动作会被不加任何处理地输送给生成的词法分析器。

5.关键字

下面是ANTLR语法中的保留字列表：import、fragment、lexer、parser、grammar、returns、locals、throws、catch、finally、mode、

options、tokens。此外，虽然rule不是一个关键字，但是避免将它作为规则或者备选分支名，因为这样会使得自动生成的RuleContext上下文对象与内置类冲突。另外，不要使用目标语言中的关键字作为词法符号、标签或者规则名。例如，if规则会生成if（）函数。

15.2 语法结构

一份语法由一个语法声明和紧随其后的若干条规则构成，具有如下的通用形式：

```
/** 可选的 Javadoc 风格的注释 */  
❶ grammar Name;  
  options {...}  
  import ... ;  
  tokens {...}  
  @actionName {...}  
  
  <<rule1>> // 可能混杂在一起的文法规则和词法规则  
  ...  
  <<ruleN>>
```

包含语法X的文件必须被命名为X.g4。其中的options、import、token声明，以及动作可以以任意次序出现。其中，option、import和token声明是可有可无的，最多只能出现一次，而位置

1

处的文件头以及至少一条规则则必须存在。规则的形式如下：

```
ruleName : <<alternative1>> | ... | <<alternativeN>> ;
```

文法规则必须以小写字母开头，词法规则必须以大写字母开头。

不带前缀的语法声明是混合语法，可以同时包含词法规则和文法规则。欲创建一份只允许文法规则出现的语法，使用如下声明：

```
parser grammar Name;  
...
```

自然，纯词法的语法如下所示：

```
lexer grammar Name;  
...
```

只有词法规则语法才能包含模式声明。

15.5节详细讲述了编写规则所需的句法。15.8节描述了语法的选项（options），15.4节给出了与语法级别的动作相关的信息。我们稍后就会看到语法的导入、词法符号声明，以及具名的动作。

1. 语法导入

如前所述，语法导入允许你将语法分解成可复用的逻辑单元。ANTLR处理被导入的语法的方式和面向对象语言中的父类非常相似。一个语法会从其导入的语法中继承所有的规则、词法符号声明和具名的动

作。位于“主语法”中的规则将会覆盖其导入的语法中的规则，以此来实现继承机制。

可以将import看作是一种智能的、不会引入本文件中已经定义过的规则的引入语句（include statement）。一系列的导入会生成一份单独的混合语法，ANTLR代码生成器看到的是最终的完整语法，即它对被导入语法毫不知情。

在处理一份主语法的过程中，ANTLR工具将所有被导入的语法加载到一起，然后将其中的规则、词法符号类型以及具名动作合并到主语法中。如图15-1所示，右侧的语法展示了导入ELang语法后的MyELang语法：

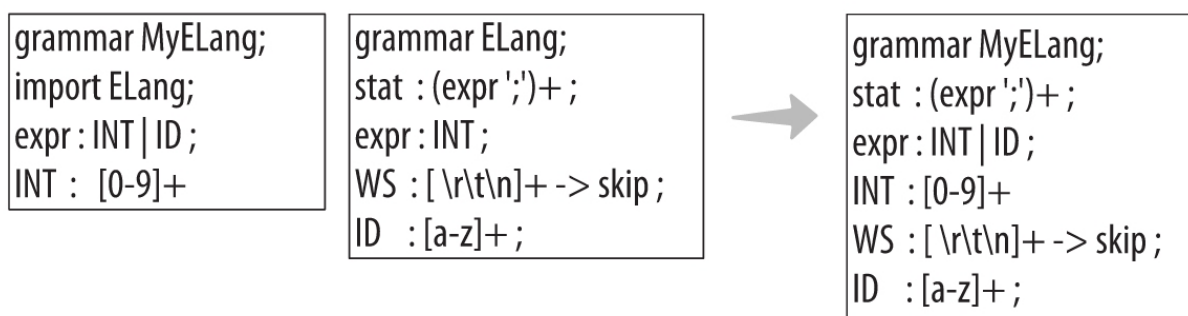


图15-1 导入ELang语法后的MyELang语法

MyELang继承了stat、WS和ID规则，并且覆盖了expr规则，增加了INT规则。下面的构建和测试步骤显示，MyELang能够识别原先的ELang所

无法识别的整数表达式。第三行的错误触发的错误消息也显示，语法分析器试图寻找的是MyELang而非ELang的expr。

```
⇒ $ antlr4 MyELang.g4
⇒ $ javac MyELang*.java
⇒ $ grun MyELang stat
⇒ 34;
⇒ a;
⇒ ;
⇒ EOF
⌞ line 3:0 extraneous input ';' expecting {<EOF>, INT, ID}
```

如果其中存在词法符号声明，主语法会将其合并到词法符号集合中。任意的具名动作，如@members同样也会被合并。通常，应当避免将具名动作放在被导入语法的规则中，因为这样会限制这些语法的复用。ANTLR会忽略被导入语法中的所有选项（options）。

被导入的语法可以导入其他语法。ANTLR按照深度优先的策略处理所有的被导入语法。如果多于一个的语法定义了规则r，ANTLR选择它发现的第一条r作为结果。在图15-2中，ANTLR按照Nested、G1、G3、G2的顺序处理语法。

Nested包含的r规则来自ANTLR首先看到的G3而非随后看到的G2。

并非每种类型的语法都能导入其他类型的语法。

- 词法语法能导入词法语法

- 句法语法能导入句法语法

·混合语法能导入词法语法或者句法语法

ANTLR将被导入的规则放置在主语法的词法规则列表末尾。这意味着，主语法中的词法规则具有比被导入语法中的规则更高的优先级。

例如，如果一份主语法定义了规则IF: 'if'，被其导入的语法定义了规则ID: [a-z]+（它也能识别if），ID不会隐藏主语法中的IF词法符号定义。

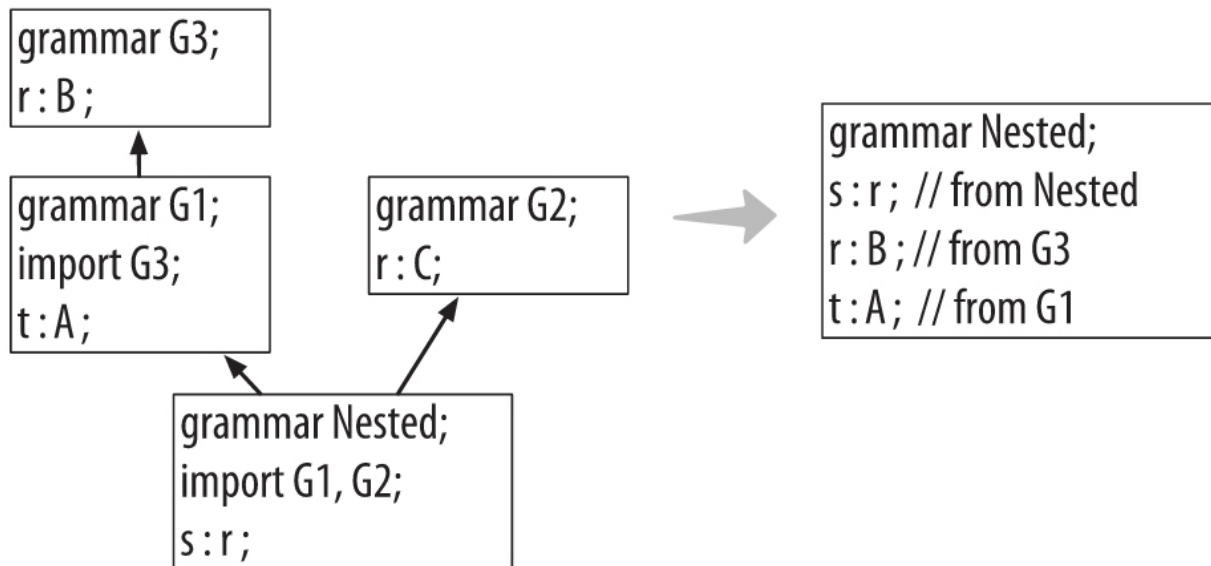


图15-2 按照Nested、G1、G3、G2顺序处理语法的ANTLR

2.词法符号声明

tokens区域存在的意义在于，它定义了一份语法所需，但却未在本语法中列出对应规则的词法符号。基本的语法如下：

```
tokens { <<Token1>>, ..., <<TokenN>> }
```


大多数情况下，`tokens`区域用于定义本语法中动作所需的词法符号类型（详见10.3节）。

```
// 显式定义关键字词法符号类型，以避免隐式定义引发的警告
tokens { BEGIN, END, IF, THEN, WHILE }
@lexer::members {    // 关键字 Map，用于在词法分析器中对词法符号赋予类型值
    Map<String,Integer> keywords = new HashMap<String,Integer>() {{
        put("begin", KeywordsParser.BEGIN);
        put("end",    KeywordsParser.END);
        ...
    }};
}
```

`tokens`区域实际上仅仅是一些会被合并到整体词法符号集合中的词法符号定义。

```
$ cat Tok.g4
grammar Tok;
tokens { A, B, C }
a : X ;
$ antlr4 Tok.g4
warning(125): Tok.g4:3:4: implicit definition of token X in parser
$ cat Tok.tokens
A=1
B=2
C=3
X=4
```

3. 语法级别的动作

10.1节中“在语法规则之外使用动作”部分展示了语法文件级别的具名动作的实际应用。当前，只有两种动作（对于Java目标语言而言）：

header和**members**。前者用于将代码注入生成的识别类中的类声明之前，后者用于将代码注入为识别类的字段和方法。

对于混合语法，**ANTLR**同时将这些代码注入到词法分析器和语法分析器中。欲令其只出现在语法分析器或者词法分析器中，使用

@parser: : name或者**@lexer: : name**。

下面的例子显示了为生成代码指定包名的过程：

```
reference/foo/Count.g4
grammar Count;

@header {
package foo;
}

@members {
int count = 0;
}

list
@after {System.out.println(count+" ints");}
      : INT {count++;} (',' INT {count++;} )*
      ;

INT : [0-9]+ ;
WS : [ \r\t\n]+ -> skip ;
```

语法本身应当位于**foo**目录中，这样**ANTLR**就会将代码生成到该目录下（也可以利用**-o**选项指定输出目录）

```

⇒ $ cd foo
⇒ $ antlr4 Count.g4 # 在当前目录 foo 中生成代码
⇒ $ ls
  Count.g4          CountLexer.java      CountParser.java
  Count.tokens      CountLexer.tokens
  CountBaseListener.java  CountListener.java
⇒ $ javac *.java
⇒ $ cd ..
⇒ $ grun foo.Count list
⇒ 9, 10, 11
⇒ EOF
  3 ints

```

Java编译器会在foo目录中寻找foo包下的类。

至此，我们已经全面了解了语法的结构，接下来让我们深入研究一下文法规则和词法规则。

15.3 文法规则

语法分析器由一系列文法规则组成，这些规则既可以位于文法语法中，也可以位于混合语法中。Java程序通过调用ANTLR自动生成的、与预期的起始规则相对应的函数来启动语法分析器。规则最基本的形式是规则名后面紧接着一个备选分支，然后是一个分号。

```

/** Javadoc 注释可以放在规则之前 */
retstat : 'return' expr ';' ;

```

规则中可以包含由|分隔的备选分支。

```

stat:  retstat
      |  'break' ';'
      |  'continue' ';'
      ;

```

备选分支是一组可以为空的规则元素列表。例如，下列规则中的空备选分支使得整条规则成为了可选的：

```

superClass
:   'extends' ID
|                                     // 空规则意味着其他的备选分支是可选的
;

```

1. 备选分支的标签

正如我们在7.4节中所看到的，我们可以使用#给最外层的备选分支添加标签，以获得更加精确的语法分析器监听器事件。一条规则中的备选分支要么全部带上标签，要么全部不带标签。下面两条规则中，备选分支都被加上了标签：

```

reference/AltLabels.g4
grammar AltLabels;
stat: 'return' e ';' # Return
     | 'break'      ';' # Break
     ;
e    : e '*' e      # Mult
     | e '+' e      # Add
     | INT          # Int
     ;

```

备选分支的标签无须位于行尾，#后的空格也不是必需的。

ANTLR为每个标签生成一个规则上下文类。例如，下面是ANTLR生成的监听器：

```
public interface AltLabelsListener extends ParseTreeListener {
    void enterMult(AltLabelsParser.MultContext ctx);
    void exitMult(AltLabelsParser.MultContext ctx);
    void enterBreak(AltLabelsParser.BreakContext ctx);
    void exitBreak(AltLabelsParser.BreakContext ctx);
    void enterReturn(AltLabelsParser.ReturnContext ctx);
    void exitReturn(AltLabelsParser.ReturnContext ctx);
    void enterAdd(AltLabelsParser.AddContext ctx);
    void exitAdd(AltLabelsParser.AddContext ctx);
    void enterInt(AltLabelsParser.IntContext ctx);
    void exitInt(AltLabelsParser.IntContext ctx);
}
```

其中，每个带标签的备选分支都对应了一个进入方法和一个退出方法。参数的类型取决于备选分支本身。你可以在不同的规则中使用相同的标签，这会使得语法分析树遍历器触发相同的事件。例如，下面是e规则的变体，它复用了标签BinaryOp：

```
e    : e '*' e      # BinaryOp
      | e '+' e      # BinaryOp
      | INT          # Int
      ;
```

ANTLR会为e生成如下的监听器事件：

```
void enterBinaryOp(AltLabelsParser.BinaryOpContext ctx);
void exitBinaryOp(AltLabelsParser.BinaryOpContext ctx);
void enterInt(AltLabelsParser.IntContext ctx);
void exitInt(AltLabelsParser.IntContext ctx);
```

如果一个备选分支名与另外一条规则名产生冲突，**ANTLR**会提示出错。下面的是规则**e**的另外一个变体，它包含两个与规则名冲突的备选分支标签：

```
reference/Conflict.g4
e    : e '*' e      # e
      | e '+' e      # Stat
      | INT          # Int
      ;
```

由规则名生成的上下文对象类的名字是通过将规则名或者备选分支标签名大写得到的，因此**Stat**标签和**stat**规则产生了冲突。

```
$ antlr4 Conflict.g4
error(124): Conflict.g4:6:23: rule alt label e conflicts with rule e
error(124): Conflict.g4:7:23: rule alt label Stat conflicts with rule stat
warning(125): Conflict.g4:2:13: implicit definition of token INT in parser
```

2.规则上下文对象

ANTLR为每个规则引用生成规则上下文对象（即语法分析树节点）的访问方法。对于只包含一条规则引用的规则，**ANTLR**会生成一个无参方法。例如，对于下列规则：

```
inc : e '++' ;
```

ANTLR生成的上下文类如下：

```
public static class IncContext extends ParserRuleContext {
    public EContext e() { ... } // 返回 e 对应的上下文对象
    ...
}
```

在规则中包含不止一个规则引用时，ANTLR也提供了对各上下文对象访问的支持。

```
field : e '.' e ;
```

ANTLR生成一个单参数方法，其参数是访问第*i*个规则元素时的索引值，另外它还生成一个方法，返回该规则对应的所有上下文对象。

```
public static class FieldContext extends ParserRuleContext {  
    public EContext e(int i) { ... }    // 获得第 i 个 e 上下文对象  
    public List<EContext> e() { ... }    // 返回所有的 e 上下文对象  
    ...  
}
```

如果另外一条规则*s*引用了*field*，那么可以通过内嵌动作来访问*field*匹配到的*e*的列表。

```
s : field  
  {  
    List<EContext> x = $field.ctx.e();  
    ...  
  }  
;
```

监听器或者访问器能够完成同样的工作。给定一个FieldContext对象*f*，*f.e*（）将会返回List<EContext>。

3.规则元素标签

可以使用=符号给规则元素增加标签，以此为规则上下文对象增加字段。

```
stat: 'return' value=e ';' # Return
    | 'break'          ';' # Break
    ;
```

这里的value就是规则e的返回值，而e是在别处定义的。

标签会成为对应的语法分析树节点类的字段。在本例中，value标签成为ReturnContext类的字段，因为Return标签的存在。

```
public static class ReturnContext extends StatContext {
    public EContext value;
    ...
}
```

你可以使用+=“列表标签（list label）”符号来方便地获取一组词法符号。例如，下列规则创建了一组Token对象，这些对象是由一个简单的数组结构匹配到的：

```
array : '{' el+=INT (',' el+=INT)* '}' ;
```

ANTLR在相应的规则上下文类中生成了一个List字段。

```
public static class ArrayContext extends ParserRuleContext {
    public List<Token> el = new ArrayList<Token>();
    ...
}
```


列表标签符号也适用于规则引用。

```
elist : exprs+=e (',' exprs+=e)* ;
```

ANTLR会生成一个List字段，用于存储这些上下文对象。

```
public static class ElistContext extends ParserRuleContext {
    public List<EContext> exprs = new ArrayList<EContext>();
    ...
}
```

4.规则元素

规则元素指明了语法分析器在特定的时间需要完成的任务，正如编程语言中的语句一样。规则元素可以是一条规则、一个词法符号或者一个字符串常量，例如expression、ID和'return'。表15-1是规则元素的所有可行值（我们稍后会详细分析动作和判定）：

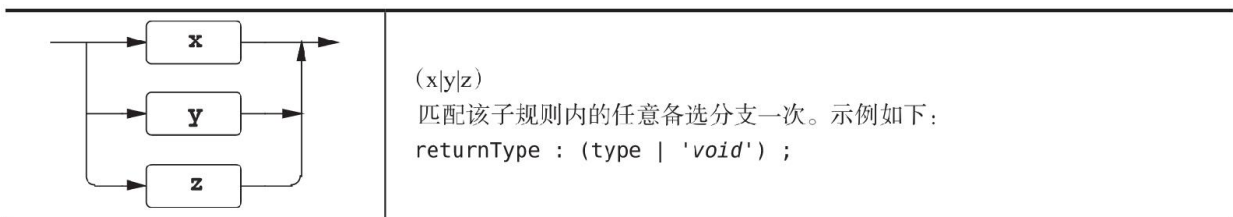
表15-1 规则元素的所有可行值

用 法	描 述
T	在当前输入位置上匹配词法符号 T。词法符号总是以大写字母开头
'literal'	在当前输入位置上匹配字符串常量。一个字符串常量就是一个由固定字符串组成的词法符号
r	在当前输入位置上匹配规则 r，这相当于像函数一样调用该规则。文法规则名总是以小写字母开头
r[«args»]	在当前输入位置上匹配 r，并且像函数调用一样传递一组参数。方括号中的参数格式依目标语言而定，通常是一组由逗号分隔的表达式列表
{«action»}	在前一个备选分支元素之后、后一个备选分支元素之前执行一段动作代码。动作中的代码符合目标语言的语法。ANTLR 原封不动地将这些动作代码拷贝到生成的类中，除了其中的属性占位符和词法符号引用，如 \$x 和 \$x.y
{«p»?}	执行语义判定 «p»。在运行时，如果 «p» 的结果为假值，那么停止对该判定之后的部分进行语法分析。判定多用在预测的场合中，当 ANTLR 需要区分多个备选分支时，它动态地开启或关闭其对应的某个（某些）备选分支
.	匹配任意除文件结束符之外的语法符号。它被称为通配符（wildcard）

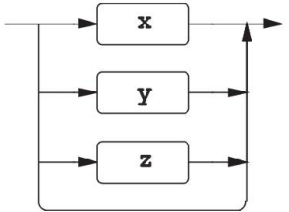
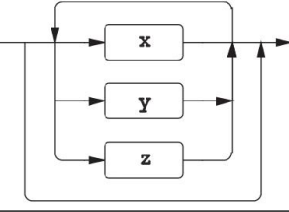
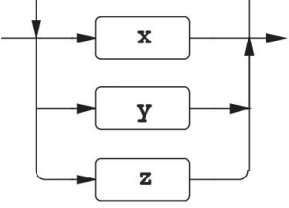
当你需要匹配除了一个/一组词法符号之外的任何东西时，使用~“非”运算符。该运算符很少被用于语法分析器中，尽管这是可行的。~INT匹配除INT之外的任意词法符号，~', '匹配除逗号之外的任意词法符号。~（INT|ID）匹配除INT或ID之外的任意词法符号。词法符号、字符串常量以及语义判定这些规则元素可以携带选项（option）。详见15.8节中的“规则元素选项”部分。

5.子规则

一条规则可以包含称为子规则的备选分支块（即扩展巴克斯-诺尔范式，Extended BNF Notation，EBNF）。子规则和规则相似，只是缺少名字并被包裹在圆括号内。在子规则的括号内，可以包含一个或者多个备选分支。子规则不能像规则一样使用local和return定义属性。存在四种类型的子规则（其中x、y、z代表语法元素）。



(续)

	<p>$(x y z)?$ 匹配子规则内的备选分支或者不匹配任何东西。示例如下：</p> <pre>classDeclaration : 'class' ID (typeParameters)? ('extends' type)? ('implements' typeList)? classBody ;</pre>
	<p>$(x y z)^*$ 匹配该子规则内的备选分支零次或多次。示例如下：</p> <pre>annotationName : ID ('.' ID)* ;</pre>
	<p>$(x y z)^+$ 匹配该子规则内的备选分支一次或多次。示例如下：</p> <pre>annotations : (annotation)+ ;</pre>

你可以在子规则中的?、*和+后加? 非贪婪运算符：??、*? 和 +?，详见15.6节。

作为简写，在子规则仅包含一个备选分支时，你可以忽略子规则两侧的括号。例如，`annotation+`和`(annotation) +`相同，`ID+`和`(ID) +`相同。简写的方式也适用于标签。`ids+=INT+`会生成一系列INT对象。

6.捕获异常

当在一条规则中发生语法错误时，ANTLR会捕获该异常，报告错误，并试图从中恢复（可能通过消费更多的词法符号来完成此过程），然后从规则中返回。每条规则都包裹在一个try/catch/finally语句中。

```

void r() throws RecognitionException {
    try {
        <<rule-body>>
    }
    catch (RecognitionException re) {
        _errHandler.reportError(this, re);
        _errHandler.recover(this, re);
    }
    finally {
        exitRule();
    }
}

```

在9.5节中，我们已经了解了使用策略对象修改ANTLR的错误处理机制的方法。但是，替换掉这种策略会影响所有的规则。欲修改单条规则的异常处理机制，可以在规则定义后指定一个异常。

```

r    :    ...
    ;
    catch[RecognitionException e] { throw e; }

```

这个例子展示了如何避免使用默认的错误报告和处理机制。r抛出的异常应当被更高层的规则所报告。指定任意异常都会令ANTLR不再生成默认的处理RecognitionException的代码。

你也可以指定其他类型的异常。

```

r    :    ...
    ;
    catch[FailedPredicateException fpe] { ... }
    catch[RecognitionException e] { ... }

```

在花括号中的代码片段以及作为“参数”的异常必须使用目标语言编写——在这个例子中就是Java。

你可将即使异常发生也需要执行的动作代码放入**finally**语句中。

```
r    :    ...  
    ;  
    // 首先是 catch 语句块  
    finally { System.out.println("exit rule r"); }
```

finally语句会在规则返回时触发的**exitRule**（）之前执行。如果需要在规则完成对备选分支的匹配之后、清理工作开始之前执行一段动作代码，你可以使用**after**。

表15-2是异常的详细列表。

表15-2 异常的详细列表

异常名	描 述
RecognitionException	这是 ANTLR 自动生成的识别器抛出的所有异常的父类。它是 <code>RuntimeException</code> 的子类，不会引起烦琐的受检异常（checked exception）。该异常记录识别器（语法分析器或者词法分析器）在当前的输入文本中所处的位置、ATN（代表语法的内部图数据结构）中的位置、规则的调用栈，以及发生错误的类型
NoViableAltException	语法分析器通过分析剩余的输入，无法决定采用多条路径中的哪一条。该异常记录了有误输入的起始词法符号，以及在错误发生时语法分析器所处的状态
LexerNoViableAltException	等价于 <code>NoViableAltException</code> ，不过仅出现在词法分析器中
InputMismatchException	当前输入的 <code>Token</code> 不符合语法分析器的预期
FailedPredicateException	在剪除不可达备选分支的预测过程中，一个语义判定执行结果为假值。预测发生在某条规则选择所采用的备选分支的过程中。如果所有的路径都被剪除了，语法分析器就会抛出 <code>NoViableAltException</code> 。如果在正常的、预测之外的语法分析过程（匹配词法符号和调用规则）中，某个判定执行结果为假值，该异常就会由语法分析器抛出

7.规则属性定义

我们需要了解许多与规则和动作相关的语法元素。规则可以像编程语言中的函数一样，包含参数、返回值以及局部变量（在规则的元素中，可以嵌入动作，我们将会在15.4节中予以学习）。ANTLR会将你定义的所有变量收集起来并存储到规则上下文对象中，这些变量通常称为属性。下面的通用形式展示了所有可行的属性定义位置：

```
rulename[«args»] returns [«retvals»] locals [«localvars»] : ... ;
```

定义在[...]中的属性的使用方法和其他任意变量一样。下面的示例规则将参数值传递给返回值：

```
// 将参数值与 INT 词法符号的值相加并返回结果  
add[int x] returns [int result] : '+' INT {$result = $x + $INT.int;} ;
```

在语法层面上，你可以指定规则级的具名动作。这样的有效命名包括 **init** 和 **after**。顾名思义，语法分析器在试图匹配相应规则之前执行 **init** 动作，在结束对相应规则的匹配之后立即执行 **after** 动作。ANTLR 的 **after** 动作不会作为自动生成的规则函数的 **finally** 代码块的一部分。可以使用 ANTLR 的 **finally** 动作来放置需要在规则函数的 **finally** 块中执行的代码。

这样的动作位于任意参数、返回值或局部变量之后。10.2节开头的 **row** 规则很好地展示了这种用法。

```
actions/CSV.g4
```

```
/** 派生自规则 "row : field (',' field)* '\r'? '\n' ;" */
row[String[] columns] returns [Map<String,String> values]
locals [int col=0]
@init {
    $values = new HashMap<String,String>();
}
@after {
    if ($values!=null && $values.size()>0) {
        System.out.println("values = "+$values);
    }
}
```

row规则接受参数columns，返回values，且定义了局部变量col。方括号中的内容将直接拷贝到生成的代码里。

```
public class CSVParser extends Parser {
    ...
    public static class RowContext extends ParserRuleContext {
        public String[] columns;
        public Map<String,String> values;
        public int col=0;
        ...
    }
    ...
}
```

生成的规则函数的参数即规则的参数，它们已经被拷贝到了局部的RowContext对象中。

```
public class CSVParser extends Parser {
    ...
    public final RowContext row(String[] columns) throws RecognitionException {
        RowContext _localctx = new RowContext(_ctx, 4, columns);
        enterRule(_localctx, RULE_row);
        ...
    }
    ...
}
```

ANTLR能够自动分析嵌套在动作中的[...], 因此String[]columns能够得到正确的解析。它也能分析出尖括号, 所以泛型参数中的逗号不会被错误解析成属性的分隔符。例如, Map<String, String>是一个属性定义。

一个动作可以包含多个属性, 即使是作为返回值的动作。在同一段动作代码中, 使用逗号分隔多个属性。

```
a[Map<String,String> x, int y] : ... ;
```

ANTLR将上述动作代码解析为两个参数x和y。

```
public final AContext a(Map<String,String> x, int y)
    throws RecognitionException
{
    AContext _localctx = new AContext(_ctx, 0, x, y);
    enterRule(_localctx, RULE_a);
    ...
}
```

8.起始规则和文件结束符

起始规则是语法分析器最初应用的规则, 它对应的规则函数被语言类应用程序所调用。例如, 一个解析Java的语言类应用程序可能会调用parser.compilationUnit (), 其中的parser是一个JavaParser对象。语法中的任意规则都可以作为起始规则。

起始规则不需要消费全部的输入文本。它们只消费能够匹配本规则的备选分支之一的、尽可能多的输入文本。例如，下列规则能够根据输入情况，自动匹配一个、两个或三个词法符号：

```
s : ID
  | ID '+'
  | ID '+' INT
  ;
```

对于`a+3`，规则`s`匹配第三个备选分支。对于`a+b`，它匹配第二个备选分支，忽略`b`。对于`a b`，它匹配第一个备选分支，忽略`b`。在后两个例子中，语法分析器并没有消费掉全部的输入文本，因为规则`s`并没有明确指明，文件结束符必须出现在匹配的备选分支之后。这种默认行为对于编写IDE之类的程序是非常有用的。想象一下，IDE试图解析某个位于巨大的Java文件中的方法。对规则`methodDeclaration`的调用应当仅匹配一个方法而忽略其后的文本。

另一方面，描述整个输入文件的规则应该引用特殊的预定义词法符号`EOF`。如果不使用它，你可能就会抓耳挠腮，感到迷惑不解：为什么任何输入都不会使起始规则报错？下列规则是一份语法的一部分，它负责读取配置文件：

```
config : element*; // 能够“匹配”带无效内容的输入文本
```

无效输入会使`config`不匹配任何输入，立即返回，且不报告错误。下面是正确的用法：

```
file : element* EOF; // 不要提前结束，必须匹配所有输入文本
```

15.4 动作和属性

在第10章中，我们已经学习了如何在语法中嵌入动作，也看到了最常见的词法符号和规则属性。本节总结了其中的重要句法和语义，提供了一份所有可用属性的完整清单。

动作是以目标语言编写的，位于花括号中的文本块，识别器根据它们在语法中的位置，在不同的时机触发之。例如，下列规则在语法分析器发现有效的定义后，打印出`found a decl`。

```
decl: type ID ';' {System.out.println("found a decl");} ;
type: 'int' | 'float' ;
```

大多数情况下，动作会访问特定词法符号和规则引用的属性。

```
decl: type ID ';'
    {System.out.println("var " + $ID.text + ": " + $type.text + ";");}
  | t=ID id=ID ';'
    {System.out.println("var " + $id.text + ": " + $t.text + ";");}
  ;
```

1.词法符号属性

所有的词法符号都包含一组预定义的只读属性。这些属性包括一些有用的数据，例如词法符号类型以及其匹配的文本。动作可以通过 `$label.attribute` 方式访问这些属性，其中 `label` 代表一个特定的词法符号（下例中的 `$a` 和 `$b`）。通常情况下，一个特定的词法符号在规则中只会出现一次，此时，在动作中，用词法符号名来引用它是不会产生歧义的。下面的例子展示了词法符号的属性表达式的用法：

```
r    :    INT {int x = $INT.line;}
        ( ID {if ($INT.line == $ID.line) ...;} )?
        a=FLOAT b=FLOAT {if ($a.line == $b.line) ...;}
    ;
```

(...) ? 子规则中的动作能够访问到自己之前的外层 `INT` 词法符号。

其中，由于存在两个 `FLOAT` 词法符号引用，动作中的 `$FLOAT` 并不是唯一的，必须使用标签来指定欲访问的词法符号引用。

在不同的备选分支中，相同的词法符号引用是唯一的，因为在任何规则的调用中，它们之中只有一个会得到匹配。例如，在下列规则中，两个备选分支的动作都可以直接引用 `$ID`，而无须使用标签。

```
r    :    ... ID {System.out.println($ID.text);}
        |    ... ID {System.out.println($ID.text);}
    ;
```

欲访问字符串常量匹配的词法符号，则必须使用标签。

```
stat:  r='return' expr ';' {System.out.println("line="+$r.line);} ;
```

大多数情况下需要访问的都是词法符号的属性，不过有时候也需要访问Token对象本身，因为它聚合起了所有的属性。此外，你可以使用它来测试一条可选的子规则是否匹配到了一个词法符号。

```
stat: 'if' expr 'then' stat (el='else' stat)?
      {if ( $el!=null ) System.out.println("found an else");}
  | ...
  ;
```

\$T和\$I分别会被解析成名为T的词法符号和标签为I的词法符号。\$ll会被解析成列表标签ll对应的List<Token>。\$T.attr会被解析成类型为T的词法符号以及表15-3中的属性attr对应的类型和值：

表15-3 属性attr对应的类型和值

属 性	类 型	描 述
text	String	词法符号匹配到的文本；它会被转换成 getText() 方法调用。例如：\$ID.text
type	int	词法符号对应的正整数类型值，如 INT。它会被转换成 getType() 方法调用。例如：\$ID.type
line	int	词法符号所处的行号，从 1 开始计数。它会被转换成 getLine() 方法调用，例如：\$ID.line
pos	int	词法符号的第一个字符在行内的位置，从 0 开始计数。它会被转换成 getCharPosition- InLine() 方法调用。例如：\$ID.pos
index	int	词法符号在词法符号流中的全局索引值，从 0 开始计数。它会被转换成 getTokenIndex() 方法调用。例如：\$ID.index
channel	int	词法符号所在的通道数。语法分析器只处理一个通道的词法符号，忽略其他通道的词法 符号。默认通道是 0（Token.DEFAULT_CHANNEL），默认隐藏通道是 Token.HIDDEN_ CHANNEL。它会被转换成 getChannel() 方法调用。例如：\$ID.channel
int	int	词法符号持有的整数值。它假设词法符号的文本是有效的数字。对于计算器之类的程序， 这个属性非常有用。它会被转换成 Integer.valueOf(text-of-token)。例如：\$INT.int

2.文法规则属性

ANTLR预定义了一系列只读的文法规则属性，供动作使用。动作只能访问自己前面的规则属性。访问名字或者标签为r的规则属性的语法是\$*r.attr*。例如，下面的\$*expr.text*返回*expr*规则匹配的全部文本的内容。

```
returnStat : 'return' expr {System.out.println("matched "+$expr.text);} ;
```

规则标签的用法如下所示：

```
returnStat : 'return' e=expr {System.out.println("matched "+$e.text);} ;
```

也可以使用\$后跟属性名来访问当前执行的规则的相应属性。例如，\$*start*是当前规则的起始词法符号。

```
returnStat : 'return' expr {System.out.println("first token "+$start.getText());} ;
```

\$*r*和\$*rl*会被解析成名为*r*或者标签名为*rl*的规则对应的ParserRuleContext对象。\$*rl*会被解析成规则列表标签名为*rl*对应的List<RuleContext>。
\$*r.attr*会被解析成表15-4中的*attr*属性对应的类型和值。

表15-4 attr属性对应的类型和值

属 性	类 型	描 述
text	String	一条规则匹配的文本或者从这条规则的起始位置到 \$text 当前位置对应的文本。需要注意的是，它包含了隐藏通道中的全部词法符号，这通常是正确的，因为其中含有全部的空白字符和注释。当作为当前规则的属性时，它可以在任何动作中使用，包括异常处理动作
start	Token	在主要词法符号通道上被规则匹配到的第一个词法符号。换句话说，该属性永远不会位于隐藏通道中。对于那些不匹配任何词法符号的规则，这个属性指向第一个后续的词法符号。当作为当前规则的属性时，它可以在规则内的任何动作中使用
stop	Token	规则匹配到的最后一个非隐藏通道的词法符号。当作为当前规则的属性时，它仅能在 after 和 finally 动作中使用
ctx	ParserRuleContext	一条规则调用对应的规则上下文对象。通过这个属性可以访问其余全部属性。例如，\$ctx.start 访问当前规则上下文对象的 start 字段。它等价于 \$start

3.动态作用域属性

和通用编程语言中的函数一样，你可以使用参数向规则传递信息，并使用返回值接收信息。但是，编程语言通常不允许其他函数访问局部变量或者参数。

例如，在Java中，从嵌套函数中访问局部变量x是非法的：

```

void f() {
    int x = 0;
    g();
}

void g() {
    h();
}

void h() {
    int y = x; // 对函数 f 中的局部变量 x 的非法引用
}

```

变量x只能在f（）的作用域中使用，该作用域是由花括号所决定的词法作用域。因此，Java的作用域是词法作用域。词法作用域是大多数编程

语言采用的方案。允许方法沿着调用链向上访问之前定义的局部变量的编程语言被称为具有动态作用域。其中的“动态”指的是编译器无法通过静态方式决定可见的变量集合。这是由于对于一个方法而言，可见变量的集合取决于它的调用者。

在语法层面上，有时候相距很远的两条规则需要进行通信，大多数情况下是为了向调用链之下的规则提供上下文信息（当然，这假设你直接在语法中嵌入了动作，而非语法分析树监听器的事件机制）。

ANTLR允许这样的动作中的动态作用域，即使用`$r: : x`访问调用者规则中的属性，其中`r`是规则名，而`x`是该规则中的属性。需要开发者自行保证，`r`是当前规则的调用者，否则，当访问`$r: : x`时，一个运行时异常会被抛出。

下面我们使用一个实际问题——保证表达式中的变量都已经被定义——来展示动态作用域的实际应用。下列语法在`block`规则中定义了`symbols`属性，并在`decl`规则中将变量名加入其中。随后的`stat`规则在这个列表中查询变量是否已经被定义。

```
reference/DynScope.g4
```

```
grammar DynScope;
```

```
prog:  block
      ;
```

```
block
```

```
/* 在这个代码块中定义的符号列表  */
```

```
locals [
```

```
    List<String> symbols = new ArrayList<String>()
```

```
]
```

```
    :   '{' decl* stat+ '}'
```

```
        // 打印 block 中的所有符号
```

```
        // $block::symbols 即作用域中定义的符号列表
```

```
        {System.out.println("symbols="+$symbols);}
```

```
    ;
```

```
/** 匹配一个变量声明，并将其标识符名加入符号列表
```

```
*/
```

```
decl:  'int' ID {$block::symbols.add($ID.text);} ';' ;
```

```
    ;
```

```
/** 匹配一个赋值语句，检查符号列表，确保赋值语句
```

```
    * 左侧的变量名已经存在。
```

```
    * contains() 方法是 List.contains(), 因为 $block::symbols
```

```
    * 是一个 List
```

```
    */
```

```
stat:  ID '=' INT ';' ;
```

```
    {
```

```
        if ( !$block::symbols.contains($ID.text) ) {
```

```
            System.err.println("undefined variable: "+$ID.text);
```

```
        }
```

```
    }
```

```
    | block
```

```
    ;
```

```
ID :  [a-z]+ ;
```

```
INT :  [0-9]+ ;
```

```
WS :  [ \t\r\n]+ -> skip ;
```

下面是构建和测试步骤的示例:


```

⇒ $ antlr4 DynScope.g4
⇒ $ javac DynScope*.java
⇒ $ grun DynScope prog
⇒ {
⇒   int i;
⇒   i = 0;
⇒   j = 3;
⇒ }
⇒ EOF
  < undefined variable: j
    symbols=[i]

```

在使用@members进行的简单字段定义和动态作用域之间存在一个重要区别。symbols是一个局部变量，所以block规则的每次调用都会生成一份拷贝。这正是我们所期望的行为，我们可以在内部代码块中复用相同的输入变量名。例如，下列嵌套代码块在内部作用域中重新定义了i。这个新定义必须隐藏掉外层作用域中的定义。

```
reference/nested-input
```

```

{
  int i;
  int j;
  i = 0;
  {
    int i;
    int x;
    x = 5;
  }
  x = 3;
}

```

下面是使用DynScope对上述输入进行语法分析的结果：

```

$ grun DynScope prog nested-input
symbols=[i, x]
undefined variable: x
symbols=[i, j]

```

`$block` : `symbols`访问最近一次调用的`block`规则上下文中的`symbols`字段。如果需要访问更上层的调用链中的`symbols`实例，可以沿着当前上下文`$ctx`，使用`getParent ()` 向上寻找。

15.5 词法规则

词法语法由词法规则组成，并且可被分解为多个模式，正如我们在12.3节“使用词法模式处理上下文相关的词法符号”部分中看到的一样。词法模式允许我们将一份词法语法分解成多个子语法。词法分析器只能返回当前模式下的规则匹配到的词法符号。

词法规则的定义方式和文法规则非常相似，除了一些例外：词法规则不能包含参数、返回值或者局部变量。词法规则名必须以大写字母开头，以和文法规则名区分开。

```
/** 可选的文档注释 */  
TokenName : «alternative1» | ... | «alternativeN» ;
```

你也可以定义一些规则，它们不是词法符号，但是却可以在识别过程中提供词法符号的功能。这样的`fragment`规则不会生成语法分析器可见的词法符号。

```
fragment HelperTokenRule : «alternative1» | ... | «alternativeN» ;
```

例如，`DIGIT`是一条非常常用的`fragment`规则。

```
INT : DIGIT+ ;           // 引用了 DIGIT 辅助规则
fragment DIGIT : [0-9] ; // 它本身不是一个词法符号
```

1.词法模式

词法模式允许你将词法规则按照上下文分组，例如XML标签内外。这和以下情况类似：多个子词法分析器负责处理业务，另外一个子词法分析器负责处理上下文。词法分析器只能返回当前模式下的规则匹配到的词法符号，词法分析器以默认模式开始。除非使用**mode**指令指定，所有的规则都处于默认模式下。模式只能出现在词法语法中，在混合语法中不允许出现（详见12.4节“将XML词法符号化”部分中的XMLLexer语法）。

```
<<rules in default mode>>
...
mode MODE1;
<<rules in MODE1>>
...
mode MODEN;
<<rules in MODEN>>
...
```

2.词法规则元素

有两种结构不能出现在文法规则中，但却可以出现在词法规则中即：..范围运算符和方括号包围的字符集合标记[characters]。不要将字符集合和文法规则中的参数相混淆。[characters]在词法分析器中仅仅意味着一个字符集合。表15-5是所有的词法规则元素的总结。

表15-5 词法规则元素总结

用 法	描 述
'literal'	匹配指定的字符或者字符序列。例如: 'while' 或者 '='
(续)	
用 法	描 述
[char set]	<p>匹配字符集中的一个字符。x-y 意为从 x 到 y 的字符集合 (包括 x 和 y)。下列转义字符会被解释为单个字符: \n、\r、\b、\t 和 \f。如果需要表达], \ 或者 -, 则必须使用 \ 进行转义。此外, 还可以使用 Unicode 字符形式: \uXXXX。下面是一些示例:</p> <p>WS : [\n\u000D] -> skip ; // 等价于 [\n\r] ID : [a-zA-Z][a-zA-Z0-9]* ; // 匹配常规的标识符 DASHBRACK : [\-\-]+ ; // 匹配 - 或者] 一次或多次</p>
'x'..'y'	匹配 x 与 y 之间的单个字符 (包括 x 和 y)。例如: 'a'..'z' 等价于 [a-z]
T	<p>调用词法规则 T。允许一般情况下的递归, 但是不允许左递归。T 可以是一个常规的词法符号或者 fragment 规则:</p> <p>ID : LETTER (LETTER '0'..'9')* ; fragment LETTER : [a-zA-Z\u0080-\u00FF] ;</p> <p>点是一个通配符, 它匹配任意的单个字符, 例如:</p> <p>ESC : '\\ ' , ; // 匹配任意 \x 转义字符</p>
{«action»}	<p>词法动作必须出现在最外层的备选分支末尾。如果一个词法规则包含多于一个备选分支, 需要将它们放置在括号中, 并令动作紧随其后:</p> <p>END : ('endif' 'end') {System.out.println("found an end");} ;</p> <p>动作代码遵循目标语言的语法。ANTLR 将动作代码拷贝到生成的代码中, 和文法规则动作不同的是, 它不会翻译形如 Sx.y 这样的表达式</p>
{«p»}?	<p>对语义判定 «p» 求值。如果 «p» 在运行时结果为假值, 其对应的规则将会变得 “不可见” (不可达)。表达式 «p» 遵循目标语言的语法。虽然语义判定可以出现在词法规则的任意位置, 但是出于效率的考虑, 最好将它们放置在规则末尾。需要注意的另外一点是, 语义判定必须出现在词法动作之前</p>
~x	<p>匹配任意不属于集合 x 的单个字符。集合 x 可以是单个字符常量、一个范围, 或者是一条形如 ~('x' 'y' 'z') 或者 ~[xyz] 的子规则。下列规则使用 ~ 和 ~[\r\n]* 匹配除指定字符外的任意字符:</p> <p>COMMENT : '#' ~[\r\n]* '\r'? '\n' -> skip ;</p>

和文法规则一样，词法规则也允许括号包围的子规则和EBNF符号的存在：？、*、+。上述COMMENT规则展示了*和？的用法。+的常见用

法是使用`[0-9]+`来匹配整数。同时，在词法规则中，也可以在上述EBNF符号后使用非贪婪后缀？。

3.递归词法规则

和大多数词法语法工具不同的是，**ANTLR**词法规则可以是递归的。这在某些情况下带来了极大的便利，例如当你希望匹配类似嵌套动作的嵌套词法符号时：`{...{...}...}`

```
reference/Recur.g4
lexer grammar Recur;

ACTION : '{' ( ACTION | ~['{}'] )* '}' ;

WS     : [ \r\t\n]+ -> skip ;
```

4.冗余字符串常量

需要注意的是，不要在多条词法规则的右侧指定相同的字符串常量。这样的字符串常量存在歧义，它将能够匹配多种类型的词法符号。

ANTLR会使这些字符串常量对语法分析器不可用。这同样适用于跨模式的规则。例如，下列词法语法定义了两个具有相同字符序列的词法符号。

```
reference/L.g4
lexer grammar L;

AND : '&' ;
mode STR;
MASK : '&' ;
```

文法语法不能引用字符串常量'&'，但是可以引用词法符号名。

```
reference/P.g4
parser grammar P;
options { tokenVocab=L; }
a : '&' // 引发一个错误：找不到词法符号
    AND // 不会引发错误
    MASK // 不会引发错误
    ;
```

下面是构建和测试步骤：

```
⇒ $ antlr4 L.g4    # 产生 P.g4 中的 tokenVocab 选项所需的 L.tokens 文件
⇒ $ antlr4 P.g4
    ⚡ error(126): P.g4:3:4: cannot create implicit token for string literal '&'
                        in non-combined grammar
```

5.词法规则动作

ANTLR词法分析器在匹配到一条词法规则后会生成一个词法符号对象。每个对词法符号的请求都从`Lexer.nextToken()`开始，该方法对每个识别到的词法符号调用`emit()`一次。`emit()`从词法分析器的当前状态收集信息，生成词法符号。它访问字段`_type`、`_text`、`_channel`、`_tokenStartCharIndex`、`_tokenStartLine`，以及`_tokenStartCharPositionInLine`。你可以通过类似`setType()`的setter方法设置这些状态。例如，若`enumIsKeyword`为假值，下列规则将`enum`转换成一个标识符：

```
ENUM : 'enum' {if (!enumIsKeyword) setType(Identifier);} ;
```

在词法规则动作中，**ANTLR**不会对特殊的**\$x**进行翻译（这一点和**ANTLR 3**不同）。一条词法规则无论包含多少个备选分支，它最多只能包含一段动作。

6.词法分析器指令

为避免语法和特定目标语言的耦合，**ANTLR**支持词法分析器指令。和任意的内嵌动作不同，这些指令遵循独立的语法，并且数量有限。词法分析器指令出现在词法规则定义的最外层备选分支末尾。每条词法符号规则最多只能包含一条指令。词法分析器指令由**->**运算符及其后的一个或多个指令名组成，可以携带参数。

TokenName : *“alternative” -> command-name*

TokenName : *“alternative” -> command-name(“identifier or integer”)*

一个备选分支可以以逗号分隔的形式携带多个指令。下面是有效的指令名：

skip 此规则不会将对应的词法符号返回给语法分析器。它常用于处理空白字符：

```
WS : [ \r\t\n]+ -> skip ;
```

more 匹配此规则，但是使用这些文本继续进行词法符号匹配。下一条词法符号规则匹配的文本将会包含当前规则匹配的文本。此指令通

常应用于模式中。下面是使用模式匹配字符串常量的一个示例：

```
reference/Strings.g4
lexer grammar Strings;
LQUOTE : '"' -> more, mode(STR) ;
WS      : [ \r\t\n]+ -> skip ;

mode STR;

STRING : '"' -> mode(DEFAULT_MODE) ; // 我们希望语法分析器看到的词法符号
TEXT   : . -> more ;                // 收集更多的文本，以生成字符串
```

下面是示例运行过程：

```
⇒ $ antlr4 Strings.g4
⇒ $ javac Strings.java
⇒ $ grun Strings tokens -tokens
⇒ "hi"
⇒ "mom"
⇒ EOF
⚡ [@0,0:3=' "hi" ',<2>,1:0]
   [@1,5:9=' "mom" ',<2>,2:0]
   [@2,11:10='<EOF>',<-1>,3:0]
```

type (T) 为当前词法符号设置类型。下列示例代码强制两个不同的词法符号使用相同的词法符号类型：

```
reference/SetType.g4
lexer grammar SetType;

tokens { STRING }

DOUBLE : '"' .*? '"' -> type(STRING) ;
SINGLE  : '\'' .*? '\'' -> type(STRING) ;
WS      : [ \r\t\n]+ -> skip ;
```

下面是示例运行过程。可以看到，两个词法符号的类型均为1。


```
⇒ $ antlr4 SetType.g4
⇒ $ javac SetType.java
```

```
⇒ $ grun SetType tokens -tokens
⇒ "double"
⇒ 'single'
⇒ EOf
  < [@0,0:7='"double"',<1>,1:0]
    [@1,9:16='single',<1>,2:0]
    [@2,18:17='<EOF>',<-1>,3:0]
```

channel (C) 为当前词法符号设置通道。默认值为 `Token.DEFAULT_CHANNEL`。你可以自行定义一些常量然后使用之，或者使用 `Token.DEFAULT_CHANNEL` 对应的整数值 `0`。另外一个名为 `Token.HIDDEN_CHANNEL` 的通用隐藏通道对应的值是 `1`。

```
@lexer::members { public static final int WHITESPACE = 1; }
...
WS : [ \t\n\r]+ -> channel(WHITESPACE) ;
```

mode (M) 在匹配当前词法符号后，将词法分析器切换到 **M** 模式。随后，词法分析器将只会使用 **M** 模式中的规则匹配词法符号。**M** 可以是一个定义于相同语法文件的模式名或者一个整数常量。参见稍早前的 `Strings` 语法。

pushMode (M) 它和 **mode** 的作用大致相同，不过它除了设置模式 **M** 之外，还会将当前模式推入一个栈中。它应当与 **popMode** 联合使用。

popMode 从模式栈中弹出一个模式，并将栈顶模式设置为当前模式。它应当与**pushMode**联合使用。

15.6 通配符与非贪婪子规则

诸如 $(...)?$ 、 $(...)*$ 和 $(...)+$ 的EBNF子规则是贪婪的 (**greedy**)——它们会消费尽可能多的输入文本，在某些情况下，这是我们所不希望看到的。在词法分析器和部分语法分析器中，类似 $.*$ 的结构会持续匹配到输入文本的末尾。若希望这样的循环是非贪婪的 (**nongreedy**)，则需要使用另外一种从正则表达式中借鉴来的语法： $. * ?$ 。我们可以通过添加 $?$ 后缀的方式使任意以 $?$ 、 $*$ 或 $+$ 结尾的子规则变成非贪婪的。非贪婪子规则在词法分析器和语法分析器中都是被允许的，不过在词法分析器中的应用更为广泛。

1.非贪婪词法子规则

下面是一条非常常见的匹配C风格注释的词法规则，它会消费所有的字符，直至遇到结尾的 $*/$ 为止：

```
COMMENT : '/*' .*? '*/' -> skip ; // .*? 匹配任意字符，直至遇到第一个 */ 为止
```

下列语法匹配允许转义双引号 \backslash "存在的字符串：

```
reference/Nongreedy.g4
grammar Nongreedy;
s : STRING+ ;
```

```

STRING : '"' ( '\\' | . ) * ? '"' ; // 匹配 "foo", "\"", "x\" \"y", ...
WS      : [ \r\t\n ] + -> skip ;

⇒ $ antlr4 Nongreedy.g4
⇒ $ javac Nongreedy*.java
⇒ $ grun Nongreedy s -tokens
⇒ "quote:\""
⇒ EOF
⚡ [ @0,0:9='\"quote:\"', <1>, 1:0]
   [ @1,11:10='<EOF>', <-1>, 2:0]

```

应当尽量少用非贪婪子规则，因为它们增加了识别的复杂度，有时甚至会使语法分析器匹配文本的过程变得棘手。词法分析器选择词法符号规则的方法如下：

- 首要目标是匹配能够识别最多输入字符的词法规则。

```

INT      : [0-9] + ;
DOT      : '.' ; // 匹配小数点
FLOAT    : [0-9] + '.' ; // '34.' 匹配此规则，而非 INT DOT

```

- 如果多于一条词法规则匹配相同的输入序列，那么在语法文件中位置靠前的规则具有更高的优先级。

```

DOC : '/*' .* ? '/' ; // 二者都可以匹配 /* foo */，胜出的是 DOC
CMT : '/*' .* ? '/' ;

```

- 非贪婪子规则匹配能够满足该规则的最少的字符序列。

```

/* 匹配双尖括号中除 \n 外的任意字符 */
STRING : '<<' ~ '\n' * ? '>>' ; // 输入 '<<foo>>>>' 的匹配结果是 STRING END
END     : '>>' ;

```

·在词法规则中非贪婪匹配子规则之后的所有决策都遵循“首先匹配”原则。

例如，`. *? ('a'|'ab')` 右侧的备选分支`'ab'`是无用代码，它将永远不会被匹配到。如果输入文本是`ab`，那么第一个备选分支`'a'`就会匹配第一个字符并结束匹配过程。与之相比，`('a'|'ab')` 本身能够使用第二个备选分支正确匹配输入`ab`。这种现象是非贪婪匹配的过程中有意为之的，目的是降低复杂度。

为展示词法规则中循环的不同用法，请看下列语法，它包含三种类似动作的词法符号（使用不同分隔符的代码共存于同一份语法中）：

reference/Actions.g4

```
ACTION1 : '{' ( STRING | . ) *? '}' ; // 允许 {"foo}
ACTION2 : '[' ( STRING | ~'"') *? ']' ; // 不允许 ["foo]; 非贪婪匹配 *?
ACTION3 : '<' ( STRING | ~[">"] ) * '>' ; // 不允许 <"foo>; 贪婪匹配 *
STRING : '"' ( '\\' | . ) *? '');
```

ACTION1规则允许不完整的字符串出现，如`{"foo}`，这是由于输入`{"foo}`匹配循环中的通配符部分，它无须进入**STRING**规则就能匹配双引号。为解决这个问题，规则**ACTION2**使用`~'"'`匹配除双引号之外的任意字符。在遇到以`]`结尾的表达式时，`~'"'`仍然是有歧义的，但是，由于其中的子规则是非贪婪的，这意味着词法分析器在遇到右方括号时就会退出循环。如果希望避免非贪婪匹配的子规则，可以显式指定一个备选分支。`~[">]`匹配除双引号和右尖括号之外的任意字符。下面是示例运行步骤：

```

⇒ $ antlr4 Actions.g4
⇒ $ javac Actions*.java
⇒ $ grun Actions tokens -tokens
⇒ {"foo}
⇒ E0F
  < [@0,0:5= '{"foo}', <1>, 1:0]
    [@1,7:6= '<EOF>', <-1>, 2:0]
⇒ $ grun Actions tokens -tokens
⇒ ["foo]
⇒ E0F
  < line 1:0 token recognition error at: '['foo]\n'
    [@0,7:6= '<EOF>', <-1>, 2:0]
⇒ $ grun Actions tokens -tokens
⇒ <"foo>
⇒ E0F
  < line 1:0 token recognition error at: '<"foo>\n'
    [@0,7:6= '<EOF>', <-1>, 2:0]

```

2.非贪婪文法子规则

在语法规则中，若语法分析器的目标是按照粗略的语法从输入文件中提取信息，即进行“模糊匹配”，非贪婪的子规则和通配符也是非常有用的。相对于词法分析器中的非贪婪决策，语法分析器总是能够作出全局性的正确决策。换句话说，语法分析器的决策不会使得合法输入在后面的某个时刻失败。非贪婪文法子规则的核心思想是：对于有效的输入序列，匹配使语法分析过程成功的最短词法符号序列。

例如，下面的语法展示了从任意Java文件中提取整数常量的方法：

```
reference/FuzzyJava.g4
```

```
grammar FuzzyJava;
/** 匹配 constant 规则匹配到的结果之间的任意文本 */
file : .*? (constant .*?)+ ;

/** 另一个更快的版本 (ANTLR 工具会对 .* 子规则给出
 * 一条警告, 可忽略之)
 */
altfile : (constant | .)* ; // 匹配一个常量或者任意词法符号 0 次或多次

/** 匹配类似 "public static final SIZE" 的文本 */
constant
    : 'public' 'static' 'final' 'int' Identifier
      {System.out.println("constant: "+$Identifier.text);}
    ;
Identifier : [a-zA-Z_$] [a-zA-Z_$0-9]* ; // 简化的标识符
```

上述语法是真实Java语法的词法规则的简单子集，整个文件大约60行。识别器仍然需要处理字符串和字符常量，以及注释，这样它才不会出错——如匹配到字符串中的整数常量。其中有一条与众不同的词法规则完成“匹配其他词法规则未能匹配的任意字符”工作。

```
reference/FuzzyJava.g4
```

```
OTHER : . -> skip ;
```

这条词法规则和语法分析器中的.*? 子规则都是成功完成模糊匹配的关键因素。

下列示例文件可以用于测试这个模糊匹配的语法分析器：

reference/C.java

```
import java.util.*;
public class C {
    public static final int A = 1;
    public static final int B = 1;
    public void foo() { }
    public static final int C = 1;
}
```

下面是构建和测试的步骤:

```
$ antlr4 FuzzyJava.g4
$ javac FuzzyJava*.java
$ grun FuzzyJava file C.java
constant: A
constant: B
constant: C
```

它忽略了除`public static final int`定义之外的全部内容。完成这些事情只需两条文法规则。

15.7 语义判定

语义判定{...}? 是使用目标语言编写的布尔表达式，它指示了沿当前判定所“守护”的路径继续进行语法分析的可行性。和动作一样，判定可以出现在文法规则的任意位置，但是只有出现在备选分支左侧的判定才具备影响分支预测（选择可行的备选分支）的能力。我们已经在第11章中详细讨论了判定。本节将对文法规则和词法规则中的语义判定进行完整的总结。下面让我们一起深入了解语法分析器在语法分析的决策过程中是如何与判定协同工作的。

1.进行带判定的语法分析决策

ANTLR的通用决策机制是在所有的可行备选分支中选择，忽略那些在当前结果为假值的判定所守护的备选分支（可行的备选分支指的是匹配当前输入的备选分支）。如果剩余的备选分支多于一个，那么语法分析器就选择在决策中位置靠前的那个。

假设有一种C++语言的变体，其中的数组引用可以用圆括号代替方括号。如果我们仅仅对其中的一个备选分支应用判定，那么`expr`规则仍会面临歧义性选择。

```
expr:          ID '(' expr ')' // 数组引用 (ANTLR 选择此条规则)
    | {istype()}? ID '(' expr ')' // 构造器风格的类型转换
    |          ID '(' expr ')' // 函数调用
    ;
```

在本例中，三个分支都可以匹配输入`x (i)`。当`x`不是类型时，判定为假值，此时`expr`中可行的备选分支只剩下了第一个和第三个。ANTLR自动选择第一个来解决歧义问题。令ANTLR自行在多个备选分支中作出选择的原因是判定的数量太少。对于`n`个可行的备选分支，最好使用至少`n-1`个判定。换句话说，不要使判定数量像`expr`一样过少。

有些时候，语法分析器会遇到与单个选择相关联的多个判定。无须担心，ANTLR能够在运行时将这些判定用合适的逻辑运算符连接起来，组成一个新的判定。

例如，在规则stat中，决策过程如下：将expr的备选分支上的判定用||连接，生成的结果用于守护stat的第二个备选分支。

```
stat:  decl | expr ;
decl:  ID ID ;
expr:  {istype()}? ID '(' expr ') ' // 构造器风格的转换
      | {isfunc()}? ID '(' expr ') ' // 函数调用
      ;
```

只有在istype () ||isfunc () 为真值的情况下，语法分析器才会在预测过程中选择stat的expr备选分支。这是十分必要的，因为语法分析器应当仅在ID是类型名或者函数名的时候才匹配一个表达式。在本例中，只进行一次判定是没有意义的。不过，请注意，当语法分析器到达expr时，它会对两个备选分支中的判定进行独立求值，以作出决策。

如果多个判定形成了一个序列，语法分析器会将它们用&&运算符连接起来。例如，考虑如下情况，在stat中的expr前增加一个判定。

```
stat:  decl | {java5}? expr ;
```

此时，仅当java5&& (istype () ||isfunc ()) 值为真时，语法分析器才会在预测过程中选择stat的第二个备选分支。

至于判定中的代码，需要将如下规则谨记于心。

(1) 使用有意义的判定

ANTLR假设你的判定用于解决歧义问题。例如，如下判定对解决两个备选分支的歧义问题毫无帮助，ANTLR会不知所措：

```
expr:  {isTuesday()}? ID '(' expr ')' // 构造器风格的转换
      |  {isHotOutside}? ID '(' expr ')' // 函数调用
      ;
```

(2) 判定不可有副作用

ANTLR假设判定可以无序执行或者求值多次，所以不要使用类似{ $\$i++ < 10$ }? 的判定。几乎可以断定，这样的判定不会按照预期方式工作。

即使不在语法分析器的决策过程中，判定也可以用于关闭备选分支，引起相应的规则失效。这发生在规则仅包含一个备选分支时。虽然只有一个选择，但是作为正常的语法分析过程，ANTLR仍会对判定求值——这和动作一样。这意味着下面的规则永远不会得到匹配：

```
prog:  {false}? 'return' INT ; // 抛出 FailedPredicateException
```

ANTLR将语法中的{false}? 转换为生成的语法分析器中的一个条件表达式。

```
if ( !false ) throw new FailedPredicateException(...);
```

迄今为止，我们看到的所有判定都是在预测过程中可见和可用的，但是也有例外。

2.寻找可见的判定

在预测过程中，语法分析器不会对动作或者词法符号引用之后的判定求值。让我们首先分析一下动作和判定之间的关系。

ANTLR对动作代码块中的内容一无所知，所以它必须假设任意判定都可能依赖动作中的副作用。设想一下，某段动作代码计算 x 的值，而另外一个判定使用了 x 。在动作创建 x 之前就对该判定求值会违背语法中指定的顺序。

更重要的是，语法分析器必须先决定匹配哪个备选分支，然后才能执行相应的动作。这是因为动作具有副作用，我们无法撤销打印输出之类的语句。例如，在下列规则中，语法分析器不能在选择该备选分支之前，执行`{java5}?`左侧的动作。

```
@members {boolean allowgoto=false;}
stat: {System.out.println("goto"); allowgoto=true;} {java5}? 'goto' ID ';'
    | ...
    ;
```

如果在预测过程中不能执行该动作，那么我们就应该对`{java5}?`求值，因为它依赖于该动作。

预测过程中也不能读取词法符号引用。读取词法符号引用具有副作用，会使得符号流向前流动一个符号。如果一个判定读取了当前的符

号，整个符号流就会失去同步。例如，在下列语法中，判定期望 `getCurrentToken()` 返回一个ID词法符号：

```
stat: '{' decl '}'  
    | '{' stat '}'  
    ;  
decl: {istype(getCurrentToken().getText())}? ID ID ';' ;  
expr: {isvar(getCurrentToken().getText())}? ID ;
```

`stat`中的决策无法对这些判定求值的原因在于，在`stat`的起始位置，当前的词法符号是一个左花括号。为了保持语义，ANTLR不会在该决策过程中对判定求值。

可见的判定（**visible predicate**）是指那些预测过程中、在动作或者词法符号之前的判定。预测过程忽略不可见的判定，把它们当作不存在。

在某些罕见情况下，语法分析器不能使用判定，即使它对于特定的决策过程是可见的。关于它们的介绍，详见下一节。

3.使用上下文相关判定

一个依赖于周围规则的参数或者局部变量的判定称为上下文相关判定（**context-dependent predicate**）。显然，我们只能在它们被定义的规则中对它们求值。例如，在下列的`prog`中对上下文相关判定`{ $S_i \leq 5$ }`？求值是没有意义的。局部变量 `S_i` 并没有在`prog`中定义。

```

prog:   vec5
      |   ...
      ;
vec5
locals [int i=1]
      :   ( {$i<=5}? INT {$i++;} ) * // 匹配 5 个 INT
      ;

```

ANTLR忽略那些无法在正确的上下文中求值的上下文相关判定。通常情况下，正确的上下文就是定义该判定的规则，但是某些时候，语法分析器即使在同一条规则中也无法对上下文相关判定求值！这些情况的检测是在运行时由自适应LL（*）预测完成的。

例如，**stat**规则中、可选的**else**分支子规则使得**stat**规则结束，语法分析器返回调用它的**prog**规则继续寻找符号：

```

prog:   stat+ ; // 允许多条连续的 stat
stat
locals [int i=0]
      :   {$i==0}? 'if' expr 'then' stat {$i=5;} ('else' stat)?
      |   'break' ';'
      ;

```

预测过程试图在if语句后寻找除else子句之外的内容。由于同一行输入中可能包含多个**stat**，**else**可选分支的预测流程重新返回了**stat**。在下一个**stat**的处理过程中，它生成了一份新的*\$i*拷贝，其值为0，而非5。此时，ANTLR忽略上下文相关判定{*\$i*==0}，因为它知道语法分析器已经不在原先的**stat**调用中了。判定过程面对的是一个不同版本的*\$i*，所以语法分析器不能对它求值。

有关词法分析器中判定的细节与之大致相同，除了一些例外：词法规则不能包含参数和局部变量。我们将在下一节详细讲述所有与词法分析器相关的内容。

4.词法规则中的判定

在文法规则中，判定必须出现在备选分支的左侧，以辅助对备选分支的预测过程。然而，在词法分析器中，判定必须出现在词法规则的右侧，因为词法分析器在看到一词法符号的全部文本后才会选择合适的规则。原则上，词法规则中的判定可以出现在规则中的任何位置。不过，某些位置会比其他位置更有效率，**ANTLR**不保证最优位置。即使在单词法符号匹配的过程中，规则中的一个判定也可能被执行多次。你可以在每条词法规则中嵌入多个判定，它们会在词法规则匹配并到达它们时被求值。

简单而言，词法分析器的目标是选择匹配最多输入字符的规则。在每个字符前，词法分析器都会检查当前还有哪些规则可用。最终，应当只有一条规则保持可用状态。此时，词法分析器就根据该规则的词法符号类型以及自己匹配到的文本，创建一个词法符号对象。

有些时候，词法分析器面对的可用规则会多于一条。例如，输入`enum`能够匹配**ENUM**规则和**ID**规则。如果`enum`的下一个字符是空格，那么二者都可以成立。词法分析器解决这样的歧义性的方法是选择位置靠

前的那条可行规则。这就是我们必须将匹配关键字的规则放置在匹配标识符的规则之前的原因：

```
ENUM : 'enum' ;  
ID   : [a-z]+ ;
```

不过，如果enum的下一个字符是一个字母，那么就只有ID是可用的了。

判定的工作原理是修改可行词法规则的集合。当词法分析器遇到一个值为假的判定时，和语法分析器一样，它会关闭该判定对应的规则。

和语法判定一样，词法判定也不能依赖于词法动作中的副作用。这是因为，动作是在词法分析器成功选定规则之后执行的，而判定是规则选择过程的一部分，它们不能依赖于动作的副作用。在词法规则中，动作必须出现在判定之后。例如，下面是另外一种在词法分析器中匹配enum关键字的方法：

reference/Enum3.g4

```
ENUM:  [a-z]+ {getText().equals("enum")}?  
        {System.out.println("enum!");}  
      ;  
ID   :  [a-z]+ {System.out.println("ID "+getText());} ;
```

ENUM中的打印动作出现在末尾，只有当前输入匹配[a-z]+和判定为真时，它才会被执行。让我们构建并测试Enum3，看看它是如何区分enum和标识符的。

```
⇒ $ antlr4 Enum3.g4
⇒ $ javac Enum3.java
⇒ $ grun Enum3 tokens
⇒ enum abc
⇒ Eof
  < enum!
    ID abc
```

它能够正常工作，不过这仅仅是出于教学目的。更加简单明了且更有效率的规则如下：

```
ENUM : 'enum' ;
```

15.8 选项

有许多语法元素和规则元素级别的选项可被设定（当前，暂时还没有规则选项）。它们能够改变ANTLR根据语法生成代码的方式。通用形式如下：

```
options { name1=value1; ... nameN=valueN; } // 与目标语言的语法无关
```

其中的**value**可以是标识符、全限定标识符（如a.b.c）、字符串、花括号包围的多行字符串，以及整数。

1. 语法选项

所有的语法都可以使用下列选项。在混合语法中，除**language**之外的所有选项都只和生成的语法分析器相关。选项的设定方式是通过在语法

文件中使用之前介绍的options，或者用ANTLR命令行的-D参数传入（详见15.9节）。下面的例子展示了这两种方法的使用，需要注意的是，-D参数会覆盖语法中的options：

superClass 设定生成的语法分析器或词法分析器的父类。对于混合语法，它设定语法分析器的父类。

```
$ cat Hi.g4
grammar Hi;
a : 'hi' ;
$ antlr4 -DsuperClass=XX Hi.g4
$ grep 'public class' HiParser.java
public class HiParser extends XX {
$ grep 'public class' HiLexer.java
public class HiLexer extends Lexer {
```

language 如果可行的话，生成指定语言的代码。否则，你会看到下列错误消息：

```
$ antlr4 -Dlanguage=C MyGrammar.g4
error(31): ANTLR cannot generate C code as of version 4.0
```

tokenVocab 在遇到文件中的词法符号时，ANTLR将词法符号的类型值赋予它们。如果需要使用不同的词法符号值，例如独立的词法分析器，可以使用此选项令ANTLR使用指定的“.tokens”文件。ANTLR会为每个语法生成一个“.tokens”文件。

```

$ cat SomeLexer.g4
lexer grammar SomeLexer;
ID : [a-z]+ ;
$ cat R.g4
parser grammar R;
options {tokenVocab=SomeLexer;}
tokens {A,B,C} // 通常，它们的类型值分别为 1,2,3
a : ID ;
$ antlr4 SomeLexer.g4
$ cat SomeLexer.tokens
ID=1
$ antlr4 R.g4
$ cat R.tokens
A=2
B=3
C=4
ID=1

```

TokenLabelType 在生成词法符号对象时，ANTLR通常使用Token类型。如果希望传给自定义的语法分析器和词法分析器一个能够生成自定义词法符号的TokenFactory，你应当将这个选项的值设为该类型。这样可以保证上下文对象清楚地知道字段和方法的返回值类型。

```

$ cat T2.g4
grammar T2;
options {TokenLabelType=MyToken;}
a : x=ID ;
$ antlr4 T2.g4

```

```

$ grep MyToken T2Parser.java
    public MyToken x;

```

2.规则选项

当前，还没有有效的规则级别的选项，不过ANTLR仍然支持下列语法，以备未来扩展：

```
rulename
options {...}
    :    ...
    ;
```

3.规则元素选项

词法符号选项的形式是T<name=value>，我们在5.4节中已经接触过了。唯一可用的词法符号选项是**assoc**，它的可行值是**left**和**right**。下面是一份示例语法，其中的左递归表达式规则指定了'^'幂运算符的词法符号选项。

```
reference/ExprLR.g4
grammar ExprLR;

expr : expr '^'<assoc=right> expr
    | expr '*' expr // 匹配乘号连接的子表达式
    | expr '+' expr // 匹配加号连接的子表达式
    | INT           // 匹配简单的整数因子
    ;

INT : '0'..'9'+ ;
WS  : [ \n]+ -> skip ;
```

语义判定也能接收选项，每个“捕获失败的语义判定”能接收一个选项。唯一可用的选项是**fail**选项，它的值可以是双引号包围的字符串常量，也可以是求值结果为字符串的动作。该字符串，或者说动作的结果字符串应当是相应判定失败后输出的消息。

```
errors/VecMsg.g4
ints[int max]
locals [int i=1]
: INT ( ',' { $i++; } { $i <= $max } ? <fail={"exceeded max " + $max}> INT ) *
;
```

当判定失败时，动作可以在执行一个函数的同时返回字符串，如：

`{...}? <fail={doSomething-AndReturnAString () }>。`

15.9 ANTLR命令行参数

如果调用ANTLR工具时没有传递命令行参数，你会看到一些帮助信息。

```
$ antlr4
ANTLR Parser Generator Version 4.0
-o ____ 指定所有的生成文件的输出位置
-lib ____ 指定语法和 tokens 文件的位置
-atn 生成规则增强转移网络图
-encoding ____ 指定语法文件的编码，例如 euc-jp
-message-format ____ 指定消息的输出风格：antlr/gun/vs2005
-listener 生成语法分析树监听器（默认行为）
-no-listener 不生成语法分析树监听器
-visitor 生成语法分析树访问器
-no-visitor 不生成语法分析树访问器（默认行为）
-package ____ 指定生成代码的包 / 命名空间
-depend 生成文件依赖
-D<option>=value 设定 / 覆盖一个语法级的选项
-Werror 将警告当作错误处理
-XdbgST 对生成的代码启动 StringTemplate 可视化器
-Xforce-atn 对所有的预测启用 ATN 模拟器
-Xlog 将详细日志保存为 antlr-timestamp.log
```

下面是这些参数的细节：

-o outdir

默认情况下，ANTLR在当前目录下生成输出文件。此参数指定ANTLR生成的语法分析器、监听器、访问器和.tokens文件的输出目录。

```
$ antlr4 -o /tmp T.g4
$ ls /tmp/T*
/tmp/T.tokens          /tmp/TListener.java
/tmp/TBaseListener.java /tmp/TParser.java
```

-lib libdir

默认情况下，ANTLR会在当前目录下寻找.tokens文件和被导入的语法。此参数指定寻找的目录。

```
$ cat /tmp/B.g4
parser grammar B;
x : ID ;
$ cat A.g4
grammar A;
import B;
s : x ;
ID : [a-z]+ ;
$ antlr4 -lib /tmp A.g4
```

-atn

此参数生成表示内部增强转移网络（Augmented Transition Network, ATN）的DOT图文件。这样的文件的文件名通常是Grammar.rule.dot。如果语法是一个混合语法，那么词法规则就会被命名为GrammarLexer.rule.dot。

```
$ cat A.g4
```

```
grammar A;
```

```
s : b ;
```

```
b : ID ;
```

```
ID : [a-z]+ ;
```

```
$ antlr4 -atn A.g4
```

```
$ ls *.dot
```

```
A.b.dot
```

```
A.s.dot
```

```
ALexer.ID.dot
```

-encoding encodingname

默认情况下，ANTLR使用UTF-8编码加载语法文件，它是一种常见编码，能够将ASCII编码成单字节。不过，世界上存在许多种编码。如果语法文件没有使用你的本地编码，那么你就需要使用此参数，使得ANTLR能够正确地读取语法文件（如下列文件）。它不影响生成的语法分析器的编码，只涉及语法文件的编码。

```
# 我的Mac OS X上的 locale 是 en_US
```

```
# 我将此文件保存为 UTF-8 编码以处理语法名：外 (\uCDE2)
```

```
$ cat 外.g4
```

```
grammar 外;
```

```
a : 'foreign' ;
```

```
$ antlr4 -encoding UTF-8 外.g4
```

```
$ ls 外*.java
```

```
外BaseListener.java 外Listener.java
```

```
外Lexer.java 外Parser.java
```

```
$ javac -encoding UTF-8 外*.java
```

-message-format format

ANTLR使用tool/resources/org/antlr/v4/tool/templates/messages/formats目录下的模板生成警告和错误消息。默认情况下，ANTLR使用antlr.stg（StringTemplate group）文件。你可以将其修改为gnu或者vs2005，从而使ANTLR生成的消息适用于Emacs或者Visual Studio。若需自定义名为X的格式，请创建一个资源org/antlr/v4/tool/templates/messages/formats/X，并将其置于CLASSPATH中。

-listener

此参数通知ANTLR生成语法分析树监听器，且是默认值。

-no-listener

此参数通知ANTLR不生成语法分析树监听器。

-visitor

默认情况下，ANTLR不生成语法分析树访问器。该参数启用此功能。ANTLR能够生成语法分析树监听器和访问器，此参数和-listener参数互斥。

-no-visitor

通知ANTLR不生成语法分析树访问器，且是默认值。

-package

使用此参数为ANTLR生成的文件指定包或者命名空间。此外，你也可以通过@header{...}动作，不过它会将语法和特定目标语言相绑定。如果你同时使用此参数和@header，需要确保header中不包含包声明，否则，生成的代码就会包含两条包声明。

-depend

不生成语法分析器和监听器，而是生成一份文件依赖列表，每行一条。输出显示被依赖和即将生成的语法。对于需要了解ANTLR语法依赖的构建工具，这是非常有用的。例如：

```
$ antlr4 -depend T.g
T.g: A.tokens
TParser.java : T.g
T.tokens : T.g
TLexer.java : T.g
TListener.java : T.g
TBaseListener.java : T.g
```

如果将-lib libdir、-depend和语法选项tokenVocab=A一起使用，那么上述依赖就会包括：T.g: libdir/A.tokens。此选项的输出目录会被-o outdir参数影响：outdir/TParser.java: T.g

-D<option>=value

使用此参数覆盖或者设定一个语法文件的语法级选项。如果需要在不修改语法的情况下生成不同语言的语法分析器，这个参数是很有用的（我期望在不远的将来，我们能够支持更多的目标语言）。

```
$ antlr4 -Dlanguage=Java T.g4 # default
$ antlr4 -Dlanguage=C T.g4
error(31): ANTLR cannot generate C code as of version 4.0
```

-Werror

在大型项目的构建中，ANTLR的警告消息很可能会被忽略。设定此参数可以令警告被当作错误，从而使得ANTLR工具能够在命令行上报告错误。

下面的一些参数主要用于调试ANTLR本身。

-XdbgST

在需要生成代码的情况下，此参数打开一个窗口，显示生成的代码以及用于生成代码的模板。它调用StringTemplate的检视器窗口。

-Xforce-atn

在允许的情况下（前瞻一个词法符号就能够作出决策，从多个备选分支中选出一个），ANTLR通常的决策方式是传统的“按词法符号类型选

择”。如果需要在这样的简单决策场景下强制使用自适应LL（*）机制，请使用此参数。

-Xlog

此参数生成一个日志文件，其中包含了大量ANTLR在处理语法的过程中生成的消息。欲了解ANTLR是如何转换左递归规则的，请使用这个参数并阅读生成的日志文件。

```
$ antlr4 -Xlog T.g4  
wrote ./antlr-2012-09-06-17.56.19.log
```

本书由“ePUBw.COM”整理，ePUBw.COM 提供
最新最全的优质电子书下载！！！！

参考文献

- [ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman, Reading, MA, Second, 2006.
- [Gro90] Josef Grosch. Efficient and Comfortable Error Recovery in Recursive Descent Parsers. *Structured Programming*. 11[3]:129–140, 1990.
- [Par09] Terence Parr. *Language Implementation Patterns*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2009.
- [Top82] Rodney W. Topor. A note on error recovery in recursive descent parsers. *SIGPLAN Notices*. 17[2]:37–40, 1982.
- [Wir78] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, Englewood Cliffs, NJ, 1978.